

5.10 Termination of a Process

The final change in process state that relates to the operation of the virtual-memory system is *exit*; this system call terminates a process, as described in Chapter 4. The part of *exit* that is discussed here is the release of the virtual-memory resources of the process. The release is done in three steps:

1. The user portions of the address space are freed, both in memory and on swap space
2. The user structure is freed
3. The page-table resources are freed

The latter two operations are somewhat complicated because the user structure and its map must be used until the process relinquishes the processor for the last time.

The first step, freeing the user address space, is identical to the one that occurs during *exec* to free the old address space. In either case, this step is not needed if the process is the child process of a *vfork*; instead, the address space is passed back to the process's parent. First, the text section of the process is released. If other processes in memory are still attached to the text, the reference count is simply decremented and the text pointer in the process structure is cleared. If there are no other users of the text image in memory, the text image is freed; it is swapped out first if it has not yet been written to swap. The image is saved for any swapped-out processes attached to the text, or for possible reattachment if the image is used again soon. The pages of the text region are freed and any pages that might be involved in page-out operations at the time are marked as *gone*, rather than their being freed immediately; the dirty-page-cleanup routine will note that the pages are gone and will place the pages on the free list when their I/O operations complete (see Section 5.12). The data and stack pages are freed similarly. The swap space is next to be freed, but this step must wait if any of the process pages were being paged out. Otherwise, the swap blocks involved might be reallocated and reused before the page-out operation occurred, destroying the new user's swap image. For this purpose, the system keeps in its proc structure a counter of the number of page-out operations in progress for each process. When any pageouts have completed, the swap blocks in the data and stack swap maps are freed.

Next the memory used by the user structure must be freed. This begins the problematical time when the process must free resources that it is not yet finished using. This condition causes two changes from the usual procedure of freeing memory. First, the page tables mapping the user structure are copied into a temporary area, so that when memory is freed with *vmemfree()*, the valid and protection bits on the page tables that map the user structure will not be cleared. Second, it would be disastrous if a page from the user structure or kernel stack were reallocated and reused before the process had finished the *exit()*. Memory is allocated either synchronously by the page-fault handler, or asynchronously by

the network from interrupt level when packets arrive (see Chapter 11). To block any allocation of memory, it is necessary to delay network interrupts by raising the processor interrupt-priority level. The process may then free the pages of its user structure, safe from having them reused until it has relinquished the processor. The next context switch will lower the priority so that interrupts may resume.

The final step in freeing the resources of the process is to free the page tables. The memory containing the page tables is freed with *vmemfree()*, using the address of the system PTEs mapping those page tables. The system PTEs are then returned to their resource map, *kernelmap*, with *rmfree()*.

With all its resources free, the exiting process finishes its manipulation of its proc entry to detach the process from its process group, and notifies its parent of its completion. The process has now become a zombie process—one with no resources, not even a kernel stack. Its parent will collect its exit status with a *wait* call, and will free its process structure.

There is nothing for the virtual-memory system to do when *wait* is called: The pages and page tables of a process are removed when *exit* is performed. On *wait*, the system just returns the process status to the caller, and deallocates the process-table entry and the small amount of space in which that resource usage information was kept.

5.11 Demand Paging

When the memory-management hardware detects an invalid PTE while translating a virtual address, it generates a trap to the system. This page-fault trap can occur for several reasons. Most 4BSD programs are created in a format that permits the executable image to be paged into main memory directly from the filesystem. When a program in a demand-paged format is first run, the kernel marks the PTEs for the text and initialized data regions of the executing process as invalid and as fill-on-demand from the filesystem. As each page of the text region is first referenced, a page fault occurs. Page faults can also occur when a process first references a page in the uninitialized data region of a program. Here, the system automatically allocates memory to the process and initializes the newly assigned page to zero. Other types of page fault arise from the simulation of reference bits and when previously resident pages have been reclaimed by the system in response to a memory shortage.

The handling of page faults centers around the *pagein()* routine; this is the routine that services all page faults. Each time *pagein()* is invoked, it is provided the virtual address that caused the fault. The first actions of *pagein()* are to find the PTE for the faulting address and to categorize the page as text, data, or stack, then to determine the state of the page. The possible states are

- Fill-on-demand
- Resident but not valid
- Paged out

Once the type and state of the page have been determined, *pagein()* must fetch the needed data, validate the PTE associated with the faulted page, and return to restart the process. The following sections describe the actions of *pagein()* for each of the possible types of pages that can be fetched.

Fill-on-Demand Pages

For a fault on a fill-on-demand page, the paging system must first determine the source of the data that the page should contain. If the page is fill-from-file, the memory free list might contain a copy of the desired page, allowing the read from disk to be avoided; see Section 5.4. If the page is found on the free list, *pagein()* reclaims it by removing it from the free list and using it for the current process. Otherwise, *pagein()* must allocate a page of main memory, fill it with the correct contents, and then record the identity of the physical page in the appropriate PTE. Zero-fill pages are simply cleared, whereas the contents of fill-from-text pages must be obtained from secondary storage. The secondary-storage location of each fill-from-text page is either specified by the disk block number and device identifier that are present in the PTE (when paging in from the filesystem), or is derived from the process swap map (for pages that have been paged out). These items and a page count fully identify the source of the disk data to be paged in. The identity of the process and the virtual address of the start of the page describe the destination of the paging operation.

To complete the page-fault servicing of a fill-on-demand page, the page needs to be marked as valid and as modified. The page is marked as modified to indicate that the swap page allocated to it does not yet have the correct contents; if the page is selected to be replaced, it will be written to swap space before being reused.

Fill-on-Demand Klustering

Servicing a fill-from-text page fault can involve significant work. Fortunately, most programs tend to exhibit some degree of *locality of reference*; that is, their memory references are localized within their virtual address space over short periods of time. If a process exhibits locality of reference, then a process's fault rate can be reduced by *prefetching* of pages that are adjacent to the one that faulted.

Whenever a fill-from-file page fault is serviced, the paging system attempts to read any pages of the file adjacent to the faulted page that may also be needed. This operation is termed *fill-on-demand klustering* and is implemented by *fodkluster()*; its name is spelled as it is to distinguish it from page clustering (described in Section 5.2). The read of the prefetched pages is piggybacked on the disk transfer that is already needed. The small additional cost of prefetching is outweighed by the improvement in service times for future page faults. Prefetched pages are placed at the tail of the free list, from which they may be reclaimed if the process uses them soon.

Kluster calculations form a set of page clusters that include a desired page and any clusters that are adjacent both in virtual memory and on the disk and are

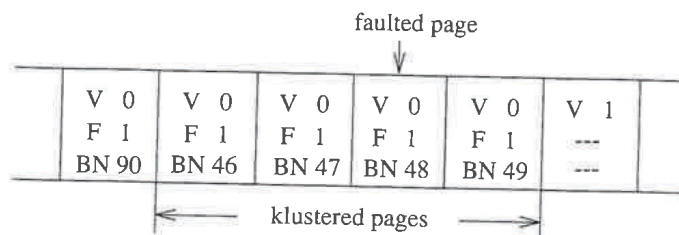


Figure 5.18 The operation of fill-on-demand klustering. V—valid bit; F—fill-on-demand bit; BN—block number on disk.

still fill-on-demand (see Fig. 5.18). No more than 16 clusters (32 pages on a VAX) will be included in a cluster set. When reading of a fill-on-demand page from the filesystem is performed, klustering will typically result in a read of an entire filesystem block (usually 4 or 8 Kbyte) instead of a single page cluster (1 Kbyte).

Interaction with the Filesystem Cache

The executable image of a program is always paged from the filesystem directly into the pages of the process, bypassing the filesystem cache. Because the filesystem uses a write-back cache (see Chapter 7), the disk blocks of an executable file might not contain the correct data soon after the file is created; instead, the data may still be in the filesystem buffer cache. There are two possible solutions to this problem. One is to check the filesystem cache **before** paging data in from the disk, copying the data from the cache if they are **present**; the other is to flush write-behind blocks from the filesystem cache before reading them in from the disk. The kernel uses the latter approach, forcing any write-behind blocks to disk before paging in data. The selection of this alternative is historic; the VAX computers on which the system was developed were CPU limited but had excess I/O capacity. Better system throughput could be obtained by having the disk controller use DMA to do the memory-to-memory copy from the buffer cache to the process text page (via the disk), than by tying up the overworked CPU doing the memory-to-memory copy. The tradeoff is reversed today, when I/O is usually the limiting factor. At an extreme is a diskless workstation; copying the data via the disk implies two transfers across a network.

Pagein of Swapped Pages

After a page has been resident in memory, it can be paged out and invalidated. Another reference to the page by the process will then generate a page fault. Here, the work to be done differs from that done for the initial fault. Whereas the first step is still to locate the PTE for the page that faulted, if the page is not fill-on-demand and the page number in the PTE is nonzero, the corresponding core-map entry must be consulted to determine the state of the page.

There are four possible states for a page that is not marked fill-on-demand, and is not valid:

1. The page was invalidated in order to sample references to it. The page number in the PTE is nonzero (valid), the core-map entry for the page is not locked, and the page is not on the free list; *pagein()* can simply mark the page as valid and return control to the process. This operation is called a *reclaim*.
2. The page has been freed but has not yet been reused. The page number in the PTE is nonzero, and the core-map entry is not locked but is on the free list. The page must be unlinked from the free list before it is validated. This operation is termed a *reclaim from free*.
3. A text page may not be resident, but another process may already have faulted the same page and started a read operation to bring the page into memory. The page number will be nonzero, and the core-map entry will be marked as locked and in transit. For pages of this type, *pagein()* must wait until the transfer finishes. The associated core-map structure is marked to indicate that some process is waiting on the page, and then *pagein()* sleeps on the address of the text structure for the page. If the page is valid when the process awakens, the *pagein* is complete. Otherwise, the page service must restart from the beginning, as the page may have changed state; the process may have been suspended for any length of time, or may even have been swapped out.
4. The page may not be resident in memory. The page may have been reused if it was freed by *pageout()* and then allocated for another purpose. For pages of this type, the page number in the PTE will have been set to zero when the page was reallocated.

If a missing page is a text page, it is possible that the page may still be in memory, in the text page cache described in Section 5.4. The page may be in memory at either of two times: when the PTE is marked fill-on-demand, and the page is still in memory from a previous invocation, or when the text page was swapped out at the time the current process attached to the text. The latter case occurs after users release a text, causing the image and page table to be swapped out and saved as a *sticky text*. 4.3BSD saves all text images for possible reuse, but previous versions of UNIX provided a *save-text* or *sticky bit* in the mode of an executable file to cause the image to be saved. When a new process attaches to the text and reloads the page tables, some of the pages are still present in memory, although the page tables contain no reference. These pages are located by a search of the free list for pages associated with the block numbers of allocated swap space. If the page is found, it will be marked *gone* because the text page table no longer holds a reference to the page. To reclaim the page, *pagein()* needs only to remove it from the free list, to place the page number in the PTE, and to mark the entry valid. This PTE must also be distributed to other processes sharing the page.

If a page that has been faulted is no longer in memory, a new page must be allocated. The usual mechanism to allocate memory cannot be used in *pagein()*

because, if the process is blocked, the state of the page might change (requiring the page-fault service to restart). Thus, *pagein()* explicitly checks for enough memory. If there is free memory, it allocates the page clusters it needs; otherwise, it sleeps on the global variable *freemem* until more memory is available, then restarts its operation.

Once memory has been allocated to hold a page that has been paged out, the remaining work is to retrieve the contents of the page from swap space. As occurs when *pagein()* is filling from the filesystem or when swapping out, adjacent pages are considered for inclusion in the transfer. Here, adjacent pages that were also swapped out are eligible for inclusion, although the size of the block considered is smaller (normally 4 Kbyte for text pages and 8 Kbyte for data pages) because the locality for data references is assumed to be larger than is the locality for text references.

If the swapped-out page is a text page, other processes sharing the text image must be able to tell that the swap has been initiated. Therefore, the changed PTEs are propagated to the other processes after the page number is filled in; this will allow them to find the core-map entry for the new page.

The memory allocator returns the core-map entry locked; *pagein()* marks the pages *in-transit*. The PTE is validated after the page has been read. For text pages, the PTE is distributed again to validate all the copies, and any processes waiting for the page are awakened. Pages that were prepaged as a result of the klustering are not validated. Instead, text pages are left reclaimable, whereas data or stack pages are freed but are left reclaimable from the free list. If the pages are used soon, they will be found in memory. Otherwise, they will be reallocated if they have not been used by the time they reach the front of the free list.

5.12 Page Replacement

The service of page faults and other demands for memory may be satisfied from the free list for some time, but eventually memory must be reclaimed for reuse. Some pages are reclaimed when processes exit. On systems with a large amount of memory and low memory demand, exiting process may provide enough free memory to fill demand. This situation arises when there is enough memory for the kernel and for all pages that have ever been used by any current process. Obviously, many computers do not have enough main memory to retain all pages in memory. Thus, it eventually becomes necessary to move some pages to secondary storage—to the swap space. Unlike paging in a page, which is demand-driven, there is no immediate indication when a page is no longer needed by a process. The kernel must implement some strategy for deciding which pages to move out of memory so that it can replace these pages with the ones that are currently needed in memory. Ideally, the strategy will choose pages for replacement that will not be needed soon. An approximation to this strategy is to find pages that have not been recently used. On the VAX, finding a page that has not been recently used is complicated by the lack of a *reference bit* in the PTEs to

mark pages that have been used. Many other virtual-memory architectures, including the Tahoe, provide such a facility.

The design of the paging system is based on results of extensive trace-driven simulation [Babaoğlu, 1981]. Accordingly, the system implements demand paging with a page-replacement algorithm that approximates global LRU. The complete per-process address space is pageable, although it maintains the familiar UNIX structure of text, data, and stack segments.

Global CLOCK Algorithm

The page-replacement policy used in 4BSD UNIX is a variant of the *CLOCK algorithm* [Corbato, 1968; Easton & Franaszek, 1979] in which reference bits are simulated in software (at least on the VAX). It is an example of a *global replacement algorithm*, one in which the choice of a page for replacement is made according to systemwide criteria; a *local replacement algorithm* would choose a process for which to replace a page, and then chose a page based on per-process criteria. The general notion of the CLOCK algorithm is to scan physical memory on a regular basis, considering pages for replacement. The part of main memory that is used by pageable processes is treated as a circular *loop* of pages, and a pointer or *clock hand* rotates around the loop one page cluster at a time. When the clock hand first reaches a cluster, the cluster is marked as unreferenced. On the next revolution, the clock hand examines the cluster to see whether the latter has been referenced in the intervening time. If the cluster has not been used, it is a strong candidate for replacement. The cluster is written to swap space if necessary, then is placed on the free list.

On a machine with a reference bit, the first pass would clear the bit and the second pass would check to see whether the reference bit had been set again. To simulate a reference bit on the VAX, 4BSD clears the valid bit on the first pass. If a process attempts to use the page that is being checked for reference (i.e., one for which the valid bit is off), the page-fault handler just sets the valid bit and returns. The second pass examines the valid bit as though it were a reference bit. The 4.3BSD Tahoe release includes revisions to the page-replacement routines to use the reference bit when it is supported by the hardware.

The simulation of reference bits on the VAX adds to the cost of operation of the CLOCK algorithm, as additional page faults occur due solely to the collection of reference information. The page-fault handler recognizes that the state of the invalid page is normal except for the simulation of the reference bit on that page. The initial cost to simulate reference bits on the VAX was about 10 percent of the total time spent in paging. On a system with a severe memory shortfall, up to 30 percent of the CPU time is spent in paging. Thus, the initial cost to simulate reference bits consumed up to 3 percent of the CPU time. To reduce the cost of reference-bit simulation, the code to handle initial page-fault characterization and revalidation was rewritten in assembly language. The result was a 70-percent drop in the time spent simulating reference bits and a reduction in the maximum simulation cost to about 1 percent of the CPU time.

The use of a single loop of pages forces all processes to compete for memory on an equal basis. This algorithm was selected based on simulations. It is interesting to note that it is also consistent with the way that UNIX treats other resources provided by the system. A common alternative to allowing all processes to compete equally for memory is to partition memory into multiple independent areas, each localized to a collection of processes that compete with one another for memory. This scheme is used, for example, by the VMS operating system [Kenah & Bate, 1984]. With this scheme, system administrators can guarantee that a process, or collection of processes, will always have a minimal percentage of memory. Unfortunately, this scheme can be difficult to administer. Setting the loop sizes too small can result in underutilization of memory and excessive I/O activity to secondary-storage devices, whereas setting them too high can result in swapping [Lazowska & Kelsey, 1978].

The 4.3BSD virtual-memory system does not use the working-set model because it lacks accurate information about the reference pattern of a process. It does track the number of pages held by a process (the *resident-set size*), but it does not know which of the resident pages constitute the working set; the count of resident pages is used only in making decisions on whether there is sufficient memory for a process to be swapped in when that process wants to run.

The system handles excessive paging by swapping out one of the processes holding the largest amounts of memory; see Section 5.13. Swapping is invoked only when paging is unable to keep up with memory needs, or when short-term resource needs warrant swapping a process. In general, the swap-scheduling mechanism does not perform well under heavy load; system performance is much better when memory scheduling can be done by the page-replacement algorithm than it is when the swap algorithm is used.

The Paging Daemon

Page replacement is done in the *pageout()* routine. When *pageout()* reclaims pages that have been modified, it is responsible for writing them to the swap area. Thus, *pageout()* must be able to use normal kernel synchronization mechanisms such as *sleep()*. It therefore runs as a separate process, the *pagedaemon*, with its own proc structure, user structure, and kernel stack. Like *init*, the *pagedaemon* is created by an internal *fork* operation during system startup (see Section 13.3); unlike *init*, however, it remains in kernel mode after the fork. The *pagedaemon* simply enters *pageout()*, which never returns. For convenience, the *pagedaemon* has a large address space into which it maps the pages that it writes to swap. Unlike other users of the disk I/O routines, the *pageout* process needs to do its disk operations asynchronously, so it can continue scanning in parallel with disk writes.

Paging Parameters

The memory-allocation needs of processes, through the page-fault handler, constantly compete with the overall system goal of maintaining a minimum threshold of pages in the free list. As the system operates, it monitors main-memory

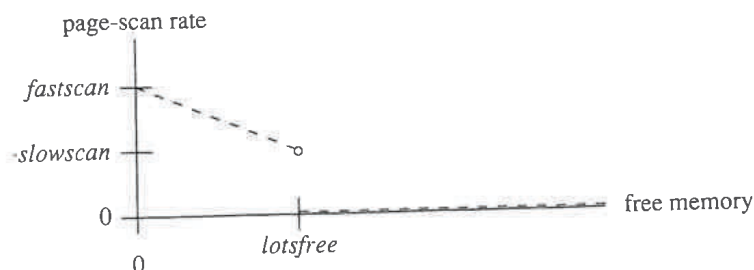


Figure 5.19 Page-scan rate.

utilization, attempting to run the *pagedaemon* frequently enough to keep the amount of free memory at or above the minimum threshold. The *pagedaemon* remains dormant until memory needs cause it to be awakened. The work of the *pagedaemon* is controlled by a number of parameters that are calculated in the *schedpaging()* routine. In general, the goal of this policy is to maintain free memory at, or above, a minimum threshold. The policy is implemented by the CLOCK algorithm, which reclaims pages for the free list. The number of pages to be reclaimed by the *pagedaemon* is a function of the memory needs of the system. As more memory is needed by the system, more pages are scanned, and the time between sampling references to pages becomes shorter. This in turn causes the number of pages freed to increase.

Four times per second, the system checks to see whether there is any need for memory. If there is, a paging rate is determined and the *pagedaemon* is awakened. The paging rate is given as the number of page clusters to scan in each quarter-second interval. *Schedpaging()* determines the need for memory by comparing the number of free memory clusters against several parameters. The first parameter, *lotsfree*, specifies a threshold (in Kbyte) for stopping the *pagedaemon*. When available memory is above this threshold, no pages will be scanned or paged out by the *pagedaemon*. *Lotsfree* is normally 512 Kbyte, and is at most one-fourth of user memory. The other interesting limits specify the desired amount of free memory that should be maintained, *desfree*, and the minimum free memory considered tolerable before involuntary swapping begins, *minfree*. *Desfree* usually is set to 200 Kbyte, and is at most one-eighth of memory; *minfree* usually is 64 Kbyte, and is at most one-sixteenth of memory. The rate at which pages are scanned for replacement varies linearly from 200 pages (or at most 20 percent of memory) per second when completely out of memory to 100 pages per second when free memory is equal to *lotsfree* (see Fig. 5.19).

The desired values for the paging parameters are communicated to the *pagedaemon* through global variables. Likewise, the *pagedaemon* records its progress, in terms of pages scanned over each time interval it runs, through a global variable.

Two-Handed Clock

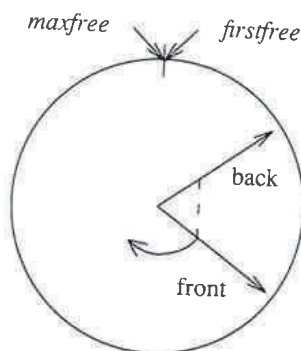
The paging parameters can be used to determine the amount of time that it takes the clock hand to revolve once around the memory. With 2 Mbyte of memory divided into 1-Kbyte clusters, the hand traverses all of memory in less than 10 seconds. On a machine with 16 Mbyte of memory, a complete revolution takes 80 seconds. The length of time allowed a process to prove its need for a page is therefore highly dependent on memory size when the normal CLOCK algorithm is used. If a sudden memory shortfall caused paging to begin after a period without scanning, the clock would require a full revolution before the first page was freed. For this reason, the CLOCK algorithm was modified in 4.3BSD to support large-memory machines more effectively. The new algorithm uses two clock hands moving through memory, with the front hand clearing reference bits and the back hand sampling the reference bits, as shown in Fig. 5.20.

The two-handed CLOCK algorithm is a generalization of the original CLOCK algorithm; in it, the distance between the hands is implicitly the size of main memory available to user processes. The reference-sampling time is proportional to the distance between the two hands and is inversely proportional to the scan rate. For simplicity, the spread between the hands is fixed at 2 Mbyte or the size of the entire memory, whichever is smaller. With the two-handed CLOCK algorithm in use, the time for sampling a page's use is independent of memory size.

Operation of Pageout

We now have enough information to examine the operation of *pageout()*, the routine that implements the page-replacement policy. *Pageout()* begins by setting the pointers for the two clock hands to be the proper distance apart. The pointers are really indices into the core map, which is treated as a circular array. The scan through memory avoids those pages used statically by the kernel. *Pageout()* then goes to sleep until free memory becomes low. It is awakened at least four times

Figure 5.20 Two-handed clock.



per second while free memory is less than *lotsfree*. Each time *pageout()* runs, it sweeps the clock hands through the core map until the desired number of page clusters have been examined, or the desired amount of free memory has been reached. At each iteration in the sweep, it examines the two page clusters, one cluster at the front hand and one at the back hand, then advances each hand by one cluster.

Clusters at the first hand are examined to determine what their state is and to see whether they are candidates for replacement. If a cluster is not free, *pageout()* invalidates the PTEs for the cluster and, if the cluster is in a shared text segment, propagates the changed PTEs to the other processes sharing the page.

When *pageout()* is examining an entry at the back hand, it checks to see whether the cluster has been used since the front hand passed. If it has been used, the valid bit will have been set again, and the cluster is left in memory. Otherwise, the cluster is selected to be freed. If any of the pages have been modified (or have not yet been written to swap), they must be "pushed" to swap before they are freed. If the pages have not been modified, they are freed immediately.

The procedure for writing process pages to the swap device, a *page push*, is somewhat complicated. The preallocated swap space for the process is described in the process's user structure. However, the push is done in the context of the *pageout* process, which has its own user structure. Therefore, a private page table (*Pushmap*) is used to map the user structure of the process owning the page into the kernel-address space.

As long as a *pageout* is needed to save the current page, adjacent pages of the process that are resident and dirty are klustered together and are written to secondary storage. If they are freed before they are next modified, they will not require separate *page-out* operations. No more than *dmmin* pages, normally 32, beginning on a *dmmin*-page boundary, are klustered together (*dmmin* is used to ensure that all the pages in such a block would be contiguous on swap space).

The mechanism used by *pageout()* to write pages to the swap area differs from normal I/O in two important ways:

1. The dirty pages are mapped into the virtual address space of the *pagedaemon*, rather than being part of the virtual address space of the process.
2. The write operation is performed asynchronously.

Both these operations are done by the *swap()* routine, which recognizes a dirty *page push* by a special flag that is passed to it. The buffer header used to describe the I/O operation is then initialized correctly for a *page push*. The PTEs for the dirty pages are copied to the *pagedaemon*'s page table; the address of the page that is to be freed is recorded in the swap buffer in the field *b_pfcnt*. Also, because *pageout()* does not sleep while the push is done, it would not normally regain control after the I/O operation completed. Therefore, *swap()* marks the buffer with a call-back flag and sets the routine for the call back to be *swdone()*. When the push completes, *swdone()* is called; it places the buffer on the list of completed *pageouts* and awakens *pageout()*.

The write being done asynchronously allows *pageout()* to continue examining pages, possibly starting additional pushes. Because the number of swap buffers is constant, care must be taken to ensure that a buffer is available before a commitment to a new page push is made. If *pageout()* has used all the swap buffers, it waits for at least one write operation to complete before it continues. As usual, blocking at this point requires that state be recomputed when *pageout()* resumes; the page at hand might have been freed or reused in the meantime. When pageout operations complete, the buffers are added to the list of completed pageouts and, if *pageout()* was blocked awaiting a buffer, it is awakened.

The list of completed pageouts is processed by *cleanup()* each time *pageout()* is awakened and just prior to another swap header being used. For each pageout operation on the list, the page cluster to be freed is placed on the memory free list unless some process has reclaimed and revalidated it. Any other pages that were included in a pageout due to klustering are unlocked, and any processes that were waiting for the locked pages are awakened. If the process using a page has abandoned that page, the *c_gone* flag will have been set in the core map, but the page will not have been freed; the page is freed at this time. Also, a count of *pageouts in progress* is kept for each process (*p_poip* in the proc entry) and for each shared text segment (*x_poip* in the text entry); this count is decremented when the pageout completes, and, if the count goes to zero, a *wakeup()* is issued. This operation is done so that exiting processes can wait for the completion of all pageout operations before freeing their references to the associated swap space.

5.13 Swapping

Although swapping is generally avoided, there are several times when it is used in 4.3BSD to address a problem not easily solved by the paging system. This problem is either a serious resource shortage, or an operation that can be most efficiently implemented with swapping. Swapping is done in 4.3BSD when any of the following occurs:

- The system becomes so short of memory that the paging process cannot free memory fast enough to satisfy the demand. For example, a memory shortfall may happen when multiple large processes are run on a machine lacking enough memory for the minimum working sets of the processes.
- The system page map becomes fragmented, preventing new or growing processes from allocating space for their page tables. Page tables do not normally shrink; however, if a process is swapped out and back in again, any extra space will be reclaimed.
- Processes are completely inactive for more than 20 seconds. Otherwise, such processes would retain a few pages of memory as well as pages for page tables.

Swapping occurs voluntarily or involuntarily. When swapping is voluntary, such as when the system page map is fragmented, it is initiated by the process that is to be swapped. When swapping is involuntary, however, it is initiated by a process called the *swapper*.

Swap operations completely remove a process from main memory, including the process page tables, the pages of the data and stack segments that are not already in swap space, the user structure, and the text segment. The text segment is handled differently from the data and stack segments because it is potentially shared by other processes.

The Swapping Process

Involuntary swapping operations are done by the swapping process, process 0. This process is the first one created by the system when it is started. The swapping policy of the *swapper* is embodied in the *sched()* routine. This routine swaps out processes when necessary, and also swaps processes back in when memory is available and they are ready to run. At any time, the swapper is in one of three states:

1. **Idle:** No swapped-out processes are ready to be run, and there is no reason to swap out anything. This is the normal state.
2. **Swapping in:** At least one runnable process is swapped out, and *sched()* attempts to find memory for it.
3. **Swapping out:** The system is short of memory or system page-table space, or there is not enough memory to swap in a process. Under these circumstances, *sched()* chooses processes and swaps them out until the memory shortage abates.

Under normal circumstances, when paging is able to keep an adequate level of free memory, the swapping process is inactive. If a process must swap itself out, or when free memory becomes low, the *swapper* is awakened.

Choosing a Process to Swap In

If more than one swapped-out process is runnable, the first task of the *swapper* is to decide which process to swap in. This decision may affect the decision whether to swap out another process. Each swapped-out process is assigned a priority based on

- The length of time it has been swapped out
- Its size when it was swapped out
- Its *nice* value, and the amount of time it was asleep since it last ran

In general, the process that has been swapped out longest or was swapped out because it was not runnable will be brought in first. Larger processes receive

lower priority, to account for the greater length of time required to swap them in and out. Having gone to the effort required to swap out a large process to free memory, it is important to keep that process out long enough to derive some benefit. Once a process is selected, the *swapper* checks to see whether there is enough memory free to swap in the process. At first, as much memory is required as was occupied by the process before it was swapped (*p_swrss*) in the process-table entry. After a process has been swapped for over 10 seconds, it will be swapped in if one-half that amount of memory is available. If there is enough memory available, the process is brought back into memory. The user structure and page tables are swapped in immediately, but the process loads the rest of its working set by demand paging from the swap device. Thus, not all the memory that was committed to the process is really used immediately. To reserve memory for processes that have been swapped in, the *swapper* creates an artificial demand for memory by setting a global variable called *deficit* to the number of pages reserved for processes that have been swapped in recently. The deficit is designed to account for the expected amount of memory that a process will need to restart; it is intended to prevent all the swapped-out processes from returning when a block of memory big enough for only one of them becomes available. The amount of memory considered to be free for the purposes of scheduling swapping and paging is therefore *freemem* minus *deficit*. As these processes will gradually bring pages into memory, *deficit* must be decreased accordingly. Rather than attempting to account for such pages brought into memory, the system reduces *deficit* gradually, by 10 percent per second, until the value reaches zero.

Involuntary Swapping

If the system does not have enough memory to swap in a process, or if it is otherwise short of memory, it will consider swapping out a process from memory. Under any of the following conditions, the system will begin to swap out processes:

- The system page map has become too fragmented to allocate page tables for some process.
- There are at least two runnable processes, the average amount of free memory has been less than that desired (*desfree*; see Section 5.12, paging parameters) over the last 5 and 30 seconds, and either the paging rate is excessive or the short-term average memory is less than *minfree*.
- A swapped-out process is ready to run, and a process is found in memory that has been sleeping for at least 20 seconds; or the swapped-out process has been out for at least 10 seconds, and the process chosen for swapping out has been in memory at least 20 seconds.

These criteria attempt to avoid swapping entirely until the *pagedaemon* is clearly unable to keep enough memory free. Once swapping has begun, an attempt is made to swap out sleeping or large, low-priority jobs. If swapping

continues to be necessary, processes that are chosen for swapping must be left out long enough to make the swap worthwhile. Finally, the processes eligible for swapping should take turns in memory, so that no process is frozen out entirely.

Many of these criteria, as well as those described in the next section, are the result of tuning on large (50 to 80 simultaneous users) timesharing systems in an academic environment.

Choosing a Process to Swap Out

When the *swapper* has committed to swapping out a process involuntarily from memory, it must choose a process to be swapped. Two criteria are used: the length of time that a process has been asleep in memory, and the size of the process. If any processes can be found that have been sleeping for more than 20 seconds (*maxslp*, the cutoff for considering the time sleeping to be "a long time"), the one sleeping for the longest time will be swapped out. Such processes have the least likelihood of making good use of the memory they occupy; thus, they are swapped out even if they are small. Otherwise, a runnable or recently run process must be swapped. The object of swapping is to free memory; thus, process size is the major criterion for choosing a process to swap. The algorithm used selects the four largest processes in memory that are eligible to be swapped. (A process may be locked into memory if it is doing raw I/O directly from its address space, or if it is doing operations on its address space, such as expanding or exiting.) Of the four processes identified, the one that has been resident in memory the longest is chosen for swapping. If free memory remains low, the system will swap out each of the four largest processes in turn, swapping another out each time one of them is swapped in. Of course, if the memory shortage abates, all the processes will be swapped in back to memory, and paging will be used to control memory use.

Swapout

Processes are swapped out of memory according to the following steps:

1. Map the user structure of the process into kernel virtual memory
2. Allocate swap space for the user structure and page tables
3. Release the text portion of the image
4. Forcibly page out all resident pages in the data and stack sections of the process
5. Write out the page tables to swap space
6. Write out the user structure and kernel stack to swap space
7. Release the user structure
8. Free the page tables

We described many of these operations in our examination of *exit* (see Section 5.10). We discussed the use of a virtual-memory window to map another process's user structure in Sections 5.7 and 5.12. For swap operations, there are two such windows, mapped by *Xswapmap* and *Xswap2map*. These windows are locked when in use, as more than one swapout could happen at the same time.

Except when a process is swapped out, its user structure and page tables remain in memory at all times. Swap space for the entire process virtual image is preallocated, but the area for the user structure and page tables is allocated only when the process is swapped out. A swapout may fail if space cannot be allocated, and therefore swap allocation is done early in the procedure. A single, contiguous block of swap space is allocated, large enough for the user structure, the data page tables, and the stack page tables. The address of the swap area is stored in the proc-structure element *p_swaddr*. The user structure is placed at the beginning of the area, followed by the page tables. Note that the page tables for the text are handled separately, as there may be at most one copy of the text page table.

When the process is eventually swapped back in, it is useful to know how much memory the process is likely to consume. Therefore, the current resident-set size of the process is recorded in the proc structure before the swap. The removal of the process then proceeds. The only subtle part of this process is related to the handling of the page tables. A process that needs to expand its page tables but is unable to do so because of fragmentation of the system page tables assumes that its page tables will be rearranged to reflect its new size when it returns to memory. Similarly, a process doing an *exec* may need to have its page tables compressed if the new size is smaller than the original one was. In both these cases, the swapping code must carefully rearrange and/or compress page tables as needed.

After the page tables have been swapped out, the user structure is written to swap space and is freed. As in *exit()*, freeing the user structure of the current process requires that the system block memory allocation by raising the processor priority until the next process-context switch. When the swapout is done by the *swapper*, no special precautions are necessary. Finally, the page-table resources for the process are freed in the same way as they are in *exit()*. If the current process is swapping itself out, the processor priority is still high enough that the page-table pages cannot be reallocated. Once the processor has switched to a new process, the priority will be lowered, and memory allocation can resume. The swapped-in process flag *SLOAD* is cleared to show that the process is not resident in memory, and if necessary the process is removed from the runnable process queue. The swapped out process cannot be run until after it is swapped back into memory.

Swapin

Processes that are **swapped out** are swapped in back to memory under the control of the *swapper*. The **procedure** is the reverse of the swapout. It is simpler in that it does not have to concern itself with pages in the user address space; instead, the

process reloads its pages on demand by paging them in. The procedure is as follows:

1. Any shared text for the process is locked against change.
2. Resources are allocated for page tables.
3. Memory is allocated for the user structure, and the user structure is read back from swap space.
4. The page tables are read into memory.
5. Swap space for the user structure and page tables is freed.
6. The text area is attached, the text page tables being swapped in if necessary.
7. The process is returned to the run queue if it is runnable.

If the process is runnable, it was removed from the run queue when it was swapped out, or was not added due to a call to *wakeup()*, as the run queue contains only runnable processes that are resident in memory. Thus, the process just swapped in must be added to the run queue if it is now runnable. The process is also marked as resident once more. If the process swapped itself out during page-table expansion, it will have marked itself with the flag SSWAP to show that it should take an alternate return when it first runs. This alternate return is necessary because the program counter saved before the swapout does not show that the swap has completed, and that the process is again ready to run. When this flag is set, the location of the alternate return (*u.u_ssave*) is marked before the SSWAP flag is cleared. When the process first runs, the *resume()* routine will arrange for a nonlocal goto (*longjmp()*).

After the swapin completes, the process is ready to run like any other, except that it has no resident pages. It will bring in the pages that it needs by faulting them, similar to operation after an *exec*, except that most of the pages are likely to be paged in from the swap device.

Swapping of Text Images

The text portion of a process being swapped in or out is handled differently from the data or stack sections, as it is potentially shared with other processes. To some degree, the text-image manipulation during swapping follows the actions during *exit* and *exec*. When a process is swapped out, it releases its use of the text image, but other processes using that image in memory may prevent it from being swapped out. A text image has two reference counts, one for the number of processes holding a reference to the text image, and the other to count the number of those processes that are currently resident in memory—the in-core reference count. When a process is swapped out, only the number of in-core references to the image is decremented. If the in-core reference count is still nonzero after this decrement, the current process needs only to be unlinked from the list of processes using the image. If the in-core count has reached zero, however, no

process using the image remains in memory, and the text image can be swapped out. The resident pages that are "modified" (have not yet been written to swap space) are pushed to the text swap area in the same way that dirty data and stack pages are swapped out. If the page tables have not yet been written to swap space, they also are swapped out; this swapout is done only once.

When a process sharing a text is swapped back into memory, the text is reattached as for an *exec*: Either the text will be in memory already and the page tables can simply be copied, or the page tables may be swapped into memory from the copy on swap space. The pages of a text image are faulted in on demand. They may be paged back from the swap space if they were resident before the text page table was swapped out; otherwise, they are paged in from the filesystem. In either case, the pages of a text image may be found in the text page cache if they have not been reused since they were freed.

Exercises

- 5.1 What does it mean for a machine to support virtual memory? What hardware facilities are typically required for a machine to support virtual memory?
- 5.2 What is the relationship between paging and swapping on a demand-paged virtual-memory system? Is it desirable to provide both mechanisms in the same system? Can you suggest an alternative to providing both mechanisms?
- 5.3 What three policies characterize paging systems? Which of these policies usually has no effect on the performance of a paging system?
- 5.4 Why is the virtual-memory system in 4.3BSD poorly suited to supporting processes with very large sparse address spaces?
- 5.5 What change in technology made it reasonable for 4.3BSD to cache more data in main memory than was done in previous systems?
- 5.6 Why does 4.3BSD simulate reference bits on PTEs on the VAX? How does it simulate them?
- 5.7 Name three functions of the core map.
- 5.8 Why were the kernel stack and user structure explicitly placed in the P1 region of the VAX virtual address space?
- 5.9 How are PTEs for pages that are to be filled from a load image identified? What information specific to this type of entry is stored in the PTE?
- 5.10 If an invalid page has a nonzero page number in its PTE, which of the following statements is certain to be true?
 - a. The page is a fill-on-demand page.

- b. The page may be in transit to the disk.
 - c. There is main memory allocated to the page.
 - d. The reference bit on the page is being simulated.
- 5.11 Why does 4.3BSD keep separate copies of a text segment's page tables, rather than just referencing a single copy?
- 5.12 Describe a disadvantage of the scheme used for the management of swap space that holds the dynamic per-process segments. *Hint:* Consider what happens when a process with a large virtual-memory size expands in small increments.
- 5.13 What are shadow swap maps? How are they used in the implementation of the *fork* system call?
- 5.14 What is *copy-on-write*? In most UNIX applications, the *fork* system call is followed almost immediately by an *exec* system call. Why does this behavior make it particularly attractive to use copy-on-write in implementing *fork*?
- 5.15 Explain why the *vfork* system call will always be more efficient than a clever implementation of the *fork* system call will be.
- 5.16 When a process expands the size of its data segment, does it actually get more main memory? What type of PTEs are created for the new pages in the virtual address space?
- 5.17 When a process exits, all its pages may not be immediately placed on the memory free list; explain why this is true.
- 5.18 The 4.3BSD virtual-memory system assumes that the overhead of memory-to-memory copy and of direct reads from the filesystem are similar. When a block to be paged in from the filesystem might be present in the filesystem buffer cache, the buffer is written to disk before the pagein. This assumption might be far from true for some current classes of virtual-memory computers. Name at least one such class. What other assumptions might be different for such computers?
- 5.19 What is klustering? Where is it used in the virtual-memory system?
- 5.20 Why was the global CLOCK algorithm modified in 4.3BSD to use two clock hands?
- 5.21 What purpose does the *pagedaemon* process serve in the virtual-memory system? What facility is used by the *pagedaemon* that is not available to a normal user process?
- 5.22 Why is the *sticky bit* no longer useful in 4.3BSD?
- 5.23 Give two reasons for swapping to be initiated.

- 5.24 Why is the current resident set size of a process recorded just before the process is swapped out?
- 5.25 Give two conditions under which a process will be swapped out when it attempts to expand its page tables.
- *5.26 Why are process's page tables never contracted except during a swap operation?
- *5.27 Consider the modifications to the global CLOCK algorithm to use two clock hands. Would it be desirable to vary the distance between the clock hands? If it would be, describe an algorithm for doing so; if it would not be, explain why not. What precautions would be necessary while the spread between the hands was being changed? How would the algorithm be affected by a limitation in the scan rate by excessive load on the paging devices?
- *5.28 What are the factors affecting the choice of software page size (or clustering factor)? On what machines would a larger page size be most efficient? On what machines would a smaller page size be more efficient?
- *5.29 Consider swap space that is spread across multiple disk partitions. What effect do widely varying partition sizes have on the swap-block interleaving scheme? Describe an alternative scheme to the one used in 4.3BSD that makes better use of disk space in this situation.
- **5.30 The 4.3BSD virtual-memory system allocates enough swap space for the full address space of a process. On some machines (such as workstations), more physical memory than swap space is available, making preallocation of swap space undesirable. Design a per-process swap map that allows swap space to be allocated as it is needed. What problems can arise from such a scheme?
- **5.31 Discuss the implementation of a virtual-memory system similar to the one in 4.3BSD on a computer with a virtual-memory architecture radically different from that of the VAX. For example, consider a machine with an "inverted page table"—a page table that has a register for each hardware page that contains the virtual address represented by the page.

References

Babaoğlu *et al.*, 1979.

Ö. Babaoğlu, W. N. Joy, & J. Porcar, "Design and Implementation of the Berkeley Virtual Memory Extensions to the UNIX Operation System," Technical Report, CS Division, EECS Dept., University of California, Berkeley, CA (December 1979).

- Babaoğlu, 1981.
Ö. Babaoğlu, "Virtual Storage Management in the Absence of Reference Bits," PhD Thesis, CS Division, EECS Dept., University of California, Berkeley, CA (1981).
- Babaoğlu & Joy, 1981.
Ö. Babaoğlu & W. N. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the Eighth Symposium on Operating Systems Principles*, pp. 78-86 (December 1981).
- Belady, 1966.
L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Systems," *IBM Systems Journal* 5(2), pp. 78-101 (1966).
- Coffman & Denning, 1973.
E. G. Coffman, Jr. & P. J. Denning, *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, NJ (1973).
- Corbato, 1968.
F. J. Corbato, "A Paging Experiment with the Multics System," Project MAC Memo MAC-M-384, Massachusetts Institute of Technology, Boston, MA (July, 1968).
- DEC, 1980.
DEC, *VAX Hardware Handbook*, Digital Equipment Corporation, Maynard, MA (1980).
- Denning, 1970.
P. J. Denning, "Virtual Memory," *Computer Surveys* 2(3), pp. 153-190 (September, 1970).
- Easton & Franaszek, 1979.
M. C. Easton & P. A. Franaszek, "Use Bit Scanning in Replacement Decisions," *IEEE Transactions on Computing* 28(2), pp. 133-141 (February, 1979).
- Gingell et al., 1987.
R. A. Gingell, J. P. Moran, & W. A. Shannon, "Virtual Memory Architecture in SunOS," *USENIX Association Conference Proceedings*, pp. 81-94 (June 1987).
- Intel, 1984.
Intel, "Introduction to the iAPX 286," Order Number 210308, Intel Corporation, Santa Clara, CA (1984).
- Joy, 1980.
W. N. Joy, "Comments on the Performance of UNIX on the VAX," Technical Report, University of California Computer System Research Group, Berkeley, CA (April 1980).
- Kashtan, 1980.
D. L. Kashtan, "UNIX and VMS: Some Performance Comparisons," Technical Report, SRI International, Menlo Park, CA (February 1980).
- Kenah & Bate, 1984.
L. J. Kenah & S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, Bedford, MA (1984).

King, 1971.

W. F. King, "Analysis of Demand Paging Algorithms," *IFIP*, pp. 485-490, North Holland (1971).

Lazowska & Kelsey, 1978.

E. D. Lazowska & J. M. Kelsey, "Notes on Tuning VAX/VMS.," Technical Report 78-12-01, Department of Computer Science, University of Washington, Seattle, WA (December 1978).

London & Reiser, 1978.

T. B. London & J. F. Reiser, "A UNIX Operating System for the DEC VAX-11/780 Computer," Technical Report TM-78-1353-4, Bell Laboratories, Murray Hill, NJ (July 1978).

Marshall, 1979.

W. T. Marshall, "A Unified Approach to the Evaluation of a Class of 'Working Set Like' Replacement Algorithms," PhD Thesis, Department of Computer Engineering, Case Western Reserve University, Cleveland, OH (May 1979).

Organick, 1975.

E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, MA (1975).

PART 3

I/O System

I/O System Overview

6.1 I/O Mapping from User to Device

Many hardware device peculiarities are hidden from the user by high-level kernel facilities, such as the filesystem (Chapter 7) and the socket interface (Chapter 10). Other such peculiarities are hidden from the bulk of the kernel itself by the I/O system. The I/O system consists of buffer caching systems, general device-driver code, and drivers for specific hardware devices that must finally address peculiarities of the specific devices. The various I/O systems are summarized in Fig. 6.1.

There are four main kinds of I/O in 4.3BSD; the *filesystem*, the *character-device* interface, the *block-device* interface, and the *socket* interface with its

Figure 6.1 Kernel I/O structure.

system-call interface to the kernel					
socket	plain file	cooked disk interface	raw disk interface	raw tty interface	cooked tty
network protocols	filesystem				line disciplines
network-interface drivers	block buffer cache				
	block device drivers		character device drivers		
the hardware					

related network devices. The character and block interfaces appear in the filesystem name space. The character interface provides *unstructured* access to the underlying hardware, whereas the block device provides *structured* access to the underlying hardware. The network devices do not appear in the filesystem, but are accessible only through the socket interface.

For the character and block devices, there is an array of entry points for the various drivers. A device is distinguished by a class and by a *device number*. The device number has two parts. The *major device number* indexes the array appropriate to the class to find entries into the appropriate device driver. The *minor device number* is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

All device drivers are connected to the rest of the kernel only by the entry points recorded in the array for their class, by their use of common buffering systems, and by their use of common low-level hardware-support routines and data structures. This segregation is important for portability, and also for configuring systems.

Character Devices

Almost all peripherals on the system except network interfaces have a character-device interface. A character device usually maps the hardware interface into a byte stream, similar to that of the UNIX filesystem. Character devices of this type include terminals (e.g., `/dev/tty00`), line printers (e.g., `/dev/lp0`), an interface to physical main memory (`/dev/mem`), and a bottomless sink for data and an endless source of end-of-file markers (`/dev/null`). Some of these, such as terminal devices, may have special behavior on line boundaries, but are still in general treated as byte streams.

Terminallike devices use *C-lists*, buffers that are smaller than those used for disks and tapes. This buffering system involves small (usually 64-byte) blocks of characters kept in linked lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for a single terminal port.

Devices such as high-speed graphics interfaces may have their own buffers or may **always** do I/O **directly** into the address space of the user; they too are classed as **character devices**. Some of these drivers may recognize special types of records, and thus be further from the plain byte-stream model.

The character interface for disks and tapes is also called the *raw device interface*; it provides an unstructured interface to the device. Its primary task is to arrange for direct I/O to and from the device. The disk driver isolates the details of tracks, cylinders, and the like from the rest of the kernel. It also handles the asynchronous nature of I/O by maintaining and ordering an active queue of pending transfers. Each entry in the queue specifies whether it is for reading or writing, what the main-memory address for the transfer is, what the device address for the transfer (usually the disk sector number) is, and what the transfer size (in bytes) is.

All other restrictions of the underlying hardware are passed through the character interface to its clients, making these character-device interfaces the furthest from the byte-stream model. Thus, the user process must abide by the sectoring restrictions imposed by the underlying hardware. For magnetic disks, the file offset and transfer size must be a multiple of the sector size. The character interface does not copy the user data into a kernel buffer before putting them on an I/O queue. It arranges to have the I/O done directly to or from the address space of the process. The size and alignment of the transfer is limited by the physical device. However, the transfer size is not restricted by the maximum size of the internal buffers of the system, since these buffers are not used. The software does restrict the size of a single transfer to 65,535 bytes; this restriction is an artifact of the PDP-11 derivation of many of the devices themselves.

The character interface is typically used by only those system utility programs that have an intimate knowledge of the data structures on the disk or tape. The character interface also allows user-level prototyping; for example, the 4.2BSD filesystem implementation was written and largely tested as a user process that used a raw disk interface before the code was moved into the kernel.

Block Devices

Block devices include disks and tapes. The task of the block-device interface is to convert from the user abstraction of a disk as an array of bytes to the structure imposed by the underlying physical medium. Although the user may wish to write a single byte to a disk, the hardware can read and write only in multiples of sectors. Hence, the system must arrange to read in the sector containing the byte to be modified, to replace the affected byte, and to write the sector back to the disk. This operation of converting random access to an array of bytes to reads and writes of disk sectors is known as *block I/O*. Block devices are accessible directly through appropriate device special files, but are more commonly accessed indirectly through the filesystem, see (Chapter 7). The kernel also uses the block device interface for disk drivers when doing swapping and paging transfers.

Processes may read data in sizes smaller than a disk block. The first time a small read is required from a particular disk block, the block will be transferred from the disk into a kernel buffer. Succeeding reads of parts of the same block then usually require only copying from the kernel buffer to the memory of the user process. Multiple small writes are treated similarly. A cache buffer is allocated when the first write to a disk block is made, and succeeding writes to the part of the same block are then likely to require only copying into the kernel buffer, and no disk I/O.

In addition to providing the abstraction of arbitrary alignment of reads and writes, the block buffer cache reduces the number of disk I/O transfers required by accesses to the filesystem. Because system parameter files, commands, and directories are read repeatedly, their data blocks are usually in the buffer cache when they are needed. Thus, the kernel does not need to read them from the disk every time that they are requested.

If the system crashes while data for a particular block are in the cache but have not yet been written to disk, the filesystem on the disk will be incorrect and those data will be lost.¹ To alleviate this problem, writes are periodically forced for dirty buffer blocks. These forced writes are done (usually every 30 seconds) by a user process, *update*, doing a *sync* system call. There is also a system call, *fsync*, that may be used to force all blocks of a single file to be written to disk immediately; this synchronization is useful for database consistency.

Most magnetic tape accesses are done, in practice, through the appropriate raw tape device, bypassing the block buffer cache. When the cache is used, tape blocks must still be written in order, so the tape driver forces synchronous writes for them.

Socket-Interface Buffering

Interprocess communication and networking use the *memory buffering system*, or simply *mbufs*. Mbufs are fully described in Chapter 10; they hold moderately sized (usually 112-byte) blocks of characters kept in linked lists. Although all free mbufs are kept in a single free list, most interprocess-communication facilities limit the number of characters that may be queued at one time for a single connection.

A *write* (or *send*) system call on a socket descriptor puts characters on an output queue for the connection. For a network connection, an initial transfer on the network often is started. For intramachine connections, if a process is waiting for data on the connection, it is made runnable or is notified that the data can be read. In either event, further activity related to delivery is handled asynchronously by the kernel, independently of user processes. If there is not enough space on the output queue for all the data supplied by a *write* system call, the system call may block until space becomes available when the network or the other process has accepted enough characters from the output queue.

Input is driven by the arrival of data from the network or from other processes on the same machine. This collecting of arriving data is done asynchronously; the user process does not have to do a *read* (or *recv*) system call first. If there is not enough space in the receiving queue when input arrives, the sending process is put to sleep until the receiver has read some of the data.

When a user process uses a *read* system call to request data from the input queue, any data in the queue are returned immediately. If there are no data in the input queue, the *read* system call may block until data are put on the queue.

6.2 Descriptor Management and Services

For user processes, all I/O is done through descriptors. The user interface to descriptors was described in Section 2.6. This section describes how the kernel

¹ Critical system data, such as the contents of directories, however, are written synchronously to disk to ensure filesystem consistency; this is described in Chapter 7.

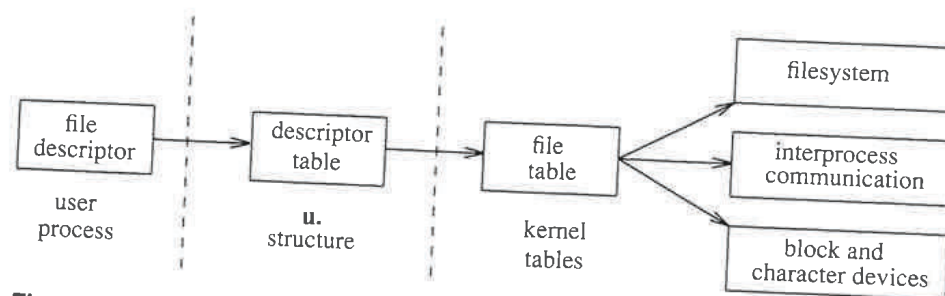


Figure 6.2 File descriptor reference to file table.

manages descriptors and how it provides descriptor services, such as locking and selecting.

System calls that refer to open files take a file descriptor as an argument to specify the file. The file descriptor is used by the kernel to index the *descriptor table* for the current process (kept in the user structure of the process) to locate a *file-table entry*, or *file structure*. The relations of these data structures are shown in Fig. 6.2.

The file-table entry provides a file type and a pointer to an underlying object for the descriptor. For data files, the file-table entry points to a table containing filesystem specific information that is described in Chapter 7. Special files do not have data blocks allocated on the disk. The kernel notices these file types and calls appropriate drivers to handle I/O for them. The 4.3BSD file-table entry may also reference a *socket* instead of a file. Sockets have a different file type, and the file-table entry points to a system block that is used in doing interprocess communication.

Open File Table

The file table is the focus of activity for file descriptors. It contains the information necessary to access the underlying objects and to maintain common information.

The file table is an object-oriented data structure. Each entry in the table contains a type and an array of function pointers that translate the generic operations on file-table descriptors into the specific actions associated with their type. In 4.3BSD, there are two descriptor types, files and sockets. The operations that must be implemented for each type are as follows:

- A function to read and write the descriptor
- A function to select on the descriptor
- A function to do *ioctl* operations on the descriptor
- A function to close, and possibly to deallocate, the object associated with the descriptor

Note that there is no open routine defined in the object table. 4.3BSD treats descriptors in an object-oriented fashion only after they are created. This was

done because sockets and files have very different characteristics, and combining these at open time would have overloaded an otherwise simple interface.

Each file-table entry has a pointer to a data structure that contains information specific to the instance of the underlying object. The data structure is opaque to the routines that manipulate the file table. A reference to the data structure is passed on each call to a function that implements a file-table operation. All state associated with an instance of an object must be stored in that instance's data structure; the underlying objects are not permitted to manipulate the file table themselves.

The *read* and *write* system calls do not take an offset in the file as an argument. Instead, each read or write updates the current *file offset* in the file according to the number of data transferred. The offset determines the position in the file for the next read or write. The offset can be set directly by the *lseek* system call. Since more than one process may open the same file, and each such process needs its own offset for the file, the offset cannot be stored in the per-object data structure. Thus, each *open* system call allocates a new file-table entry, and the open file table contains the offset.

Semantics associated with all file descriptors are enforced at the descriptor level before the system calls through the function entries. These semantics are maintained in a set of flags associated with the descriptor. For example, the flags record whether the descriptor is open for **reading and/or writing**. If a descriptor is marked as open only for reading, an **attempt to write it** will be caught by the descriptor code. Thus, the functions **defined for doing reading and writing** need not check the validity of the request; we can implement them knowing that they will never receive an invalid request.

Other information maintained in the flags includes

- The *no-delay* (NDELAY) flag. If a read or a write would cause the process to block, the system call returns an error (EWOULDBLOCK) instead.
- The *asynchronous* (ASYNC) flag. The kernel watches for a change in the status of the descriptor and arranges to send a signal (SIGIO) when a read or write becomes possible.

Other information that is specific to regular files also is maintained in the flags field:

- Information on whether the descriptor holds a shared or exclusive lock on the underlying file. Although the locking primitives could be extended to work on sockets as well as files, the descriptors for a socket usually refer to the same file-table entry.
- The *append* flag. Each time a write is made to the file, the offset pointer is first set to the end of the file. This feature is useful when, for example, multiple processes are writing to the same log file.

Each entry in the file table has a reference count. A single process may have multiple references to the entry because of calls to the *dup* or *fcntl* system calls. Also,

file structures are inherited by the child process after a *fork*, so several different processes may reference the same file-table entry. Thus, a read or write by either process on the twin descriptors will advance the file offset. This allows two processes to read the same file or to interleave output to the same file alternately, while another process that has independently opened the file will refer to that file through a different file structure with a different file offset. This capability was the original reason for the existence of the file structure; the file structure provides a place for the file offset intermediate between the descriptor and the underlying object.

Each time a new reference is created, the reference count is incremented. When a descriptor is closed (either explicitly with a *close*, or implicitly after an *exec* because the descriptor has been marked as close-on-exec, or on exit), the reference count is decremented. When the reference count finally drops to zero, the file-table entry can be freed.

The UNIX domain interprocess-communication facility allows descriptors to be sent between processes. While a descriptor is in transit between processes, it may not have any explicit references. It must not be deallocated as it will be needed when the message is received by the destination process. However, the message might never be received; thus, the file table also holds a *message count* for each entry. The message count is incremented for each descriptor that is in transit, and is decremented when the descriptor is received. The file-table entry might need to be reclaimed when all the remaining references are in messages. For more details on message passing in the UNIX domain, see Section 10.6.

The close-on-exec flag is kept in the descriptor table rather than in the file table. This flag is not shared among all the references to the file table, as it is an attribute of the file descriptor itself. The close-on-exec flag is the only piece of information that is kept in the descriptor table rather than being shared in the file table.

Management of Descriptors

The *fcntl* system call manipulates the file structure. It can be used to make the following changes to a descriptor:

- Duplicate a descriptor as if by a *dup* system call.
- Get or set the close-on-exec flag. When a process *forks*, all the parent's descriptors are duplicated in the child. The child process then *execs* a new process. Any of the child's descriptors that were marked close-on-exec are closed. The remaining descriptors are available to the newly executed process.
- Set the descriptor into nonblocking mode. If any data are available for a read operation, or any space is available for a write operation, an immediate partial read or write is done. If no data are available for a read operation, or a write operation would block, the system call returns an error showing that the operation would block, instead of putting the process to sleep. This facility was never implemented for regular files in 4.3BSD.

- Force all writes to append data to the end of the file, instead of at the descriptor's current location in the file.
- Send a signal to the process when it is possible to do I/O.
- Send a signal to a process when an exception condition arises, such as when urgent data arrive on an interprocess-communication channel.
- Set or get the process identifier or process-group identifier to which the two I/O related signals in the previous steps should be sent.

The implementation of the *dup* system call is easy. The kernel scans the descriptor table for the current process, starting at descriptor zero, until it finds an unused entry. If no unused entries exist, the kernel returns an error. Otherwise, the kernel allocates the entry to point to the same file-table entry as does the descriptor it has been asked to duplicate. The kernel then increments the reference count on the file-table entry, and returns the index of the allocated descriptor-table entry. The *fcntl* system call provides a similar function, except that it specifies a descriptor from which to start the scan.

Sometimes, a process wants to allocate a specific descriptor-table entry. Such a request is made with *dup2*. The process specifies the descriptor-table index into which the duplicated reference should be placed. The kernel implementation is the same as for *dup*, except that the scan to find a free entry is changed to close the requested entry if it is open, and then to allocate it as before. No action is taken if the new and old descriptors are the same.

The system implements getting or setting the close-on-exec flag via the *fcntl* system call by doing the appropriate action to the flags field of the associated descriptor-table entry. Other attributes that *fcntl* can get or set manipulate the flags in the file table. However, the implementation of the various flags cannot be handled by the generic code that manages the file table. Instead, the file-table flags must be passed through the object interface to the type-specific routines to do the appropriate operation on the underlying object. For example, manipulation of the nonblocking flag for a socket must be done by the socket layer, since the flag resides in a socket structure and the socket structure is opaque to the routines that operate on the file table.

The implementation of the *ioctl* system call is broken into two major parts. The upper level handles the system call itself. The *ioctl* call includes a descriptor, a command, and pointer to a data area. The command argument encodes the size of the data area for the parameters and whether the parameters are input and/or output. The upper level is responsible for decoding the command argument, allocating a buffer, and copying in any input data. If a return value is to be generated and there is no input, the buffer is zeroed. Finally, the *ioctl* is dispatched through the file-table *ioctl* function, along with the I/O buffer, to the lower-level routine that implements the requested operation.

The lower level does the requested operation. Along with the command argument, it receives a pointer to the I/O buffer. The upper level has already checked for valid memory references, but the lower level must do more precise

argument validation because it knows more about the expected nature of the arguments. However, it does not need to copy the arguments in or out of the user process. If the command is successful and produces output, the lower level places the results in the buffer provided by the top level. When the lower level returns, the upper level copies the results to the process.

Descriptor Locking

Early UNIX systems had no provision for locking files. Processes that needed to synchronize the updates of a file had to use a separate *lock file*. A process would try to create a *lock file*. If the creation succeeded, then the process could proceed with its update; if the creation failed, the process would wait and then try again. This mechanism had three drawbacks:

1. Processes consumed CPU time by looping over attempts to create locks.
2. Locks left lying around because of system crashes had to be manually removed (normally in a system startup command script).
3. Processes running as the special system-administrator user, the *superuser*, are always permitted to create files, and so were forced to use a different mechanism.

Although it is possible to get around all these problems, the solutions are not straightforward, so a mechanism for locking files was added in 4.2BSD.

The most general locking schemes allow multiple processes to update a file concurrently. Several of these techniques are discussed in [Peterson, 1983]. A simpler technique is to serialize access to a file with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking facilities for UNIX that support a byte-level granularity are well understood [Bass, 1981] and even appear in System V. However, for the standard system applications, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes can be classified according to the extent that they are enforced. A scheme in which locks are enforced for every process without choice is said to use *mandatory locks*, whereas a scheme in which locks are enforced for only those processes that request them is said to use *advisory locks*. Clearly, advisory locks are effective only when all programs accessing a file use the locking scheme. With mandatory locks, there must be some override policy implemented in the kernel. With advisory locks, the policy is left to the user programs. In the UNIX system, programs with superuser privilege are allowed to override any protection scheme. Because many of the programs that need to use locks must also run as the superuser, 4.2BSD implemented advisory locks rather than creating an additional protection scheme that was inconsistent with the UNIX philosophy or that could not be used by privileged programs.

The 4.2BSD file-locking facilities allow cooperating programs to apply advisory *shared* or *exclusive* locks on files. Only one process may have an exclusive lock on a file, whereas multiple shared locks may be present. Both

shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the lock request will block until the lock can be obtained. Because shared and exclusive locks are only advisory, even if a process has obtained a lock on a file, another process may access the file if it ignores the locking mechanism.

Locks are applied or removed on only open files. Thus, a process can manipulate locks without needing to close and reopen a file. This feature is useful, for example, when a process wishes to apply a shared lock, read some information, determine whether an update is required, then apply an exclusive lock and update the file.

A request for a lock will cause a process to block if the lock cannot be obtained immediately. In certain instances, this is unsatisfactory. For example, a process that wants only to check whether a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock cannot be obtained immediately. Being able to request a lock conditionally is useful to *daemon* processes that wish to service a spooling area. If the first instance of the *daemon* locks the directory where spooling takes place, later *daemon* processes can easily check to see whether an active *daemon* exists. Since locks exist only while the locking processes exist, lock files can never be left active after the processes exit or if the system crashes.

Implementation of Locking

The implementation of advisory locks is shown in Fig. 6.3. A process requesting either type of lock must first wait for any exclusive locks to clear. Whenever the kernel is about to put a process to sleep, the kernel must first check to see whether it is processing a nonblocking request. For nonblocking requests, the kernel must return an error, rather than putting the process to sleep. If the process wants a shared lock, it can proceed. If the process wants an exclusive lock, it must wait for any shared locks to clear. Since several processes may want an exclusive lock, they may all be waiting at line 20 in Fig. 6.3. Only one of them can get the lock; the rest must wait. Since all processes waiting at line 20 are awakened when the shared lock clears, they must all start over at the first line of Fig. 6.3. One process will make it through the code and acquire the lock; the rest will be blocked at line 9.

The processes waiting at lines 9 and 20 are awakened when other processes release their locks. When the last shared lock is released, all processes waiting for the shared lock to clear (line 20) are started. When an exclusive lock is released, all processes waiting for the exclusive lock to clear (line 9) are started.

Any resource-allocation scheme with two or more resources that allow exclusive use can create an allocation deadlock. In general, detection of deadlock is difficult [Shaw, 1974]. The 4.3BSD locking code makes a cursory attempt at preventing deadlock. The code at lines 3 through 5 ensures that a process seeking a shared lock will not deadlock against an exclusive lock that it already holds.


```

1   top:
2   while (an exclusive lock exists) {
3       if (current process has an exclusive lock) {
4           free its exclusive lock
5       }
6       if (any process holds an exclusive lock) {
7           if (nonblocking request)
8               return EWOULDBLOCK
9           wait for exclusive lock to clear
10      }
11  }
12  if (current process wants an exclusive lock) &&
13      (any process holds a shared lock) {
14      if (current process has a shared lock) {
15          free its shared lock
16      }
17      if (any process holds a shared lock) {
18          if (nonblocking request)
19              return EWOULDBLOCK
20          wait for shared lock to clear
21      }
22      goto top
23  }
24  acquire requested lock type

```

Figure 6.3 Algorithm to acquire a lock.

The code at lines 14 through 16 ensures that a process seeking an exclusive lock will not deadlock against a lock that it already holds.

The decision not to put deadlock detection into the kernel is pragmatic. Doing deadlock detection in all situations is complex and time consuming [Kenah & Bate, 1984]. The locking mechanism is designed to be fast and simple. Most locks (mailbox locks, for example) are used on only a single resource at a time, and consequently have no possibility of deadlock. More complex applications can more efficiently develop their own specialized strategies for doing deadlock detection, rather than depending on a slow generalized mechanism that is implemented in the kernel.

Multiplexing I/O on Descriptors

A process sometimes wants to handle I/O on more than one descriptor. For example, a remote login program wants to read data from the keyboard and send them through a socket to a remote machine. The program also wants to read data from the socket connected to the remote end and to write them to the screen. If a process makes a read request when there are no data available, it is normally blocked

in the kernel until the data become available. In our example, blocking is unacceptable. If the process reads from the keyboard and blocks, it will be unable to read data from the remote end that are destined for the screen. The user cannot type any input until more data have arrived from the remote end; hence, the session deadlocks. Conversely, if the process reads from the remote end when there are no data for the screen, it will block and will be unable to read from the terminal. Again, deadlock would occur if the remote end were waiting for output before sending any data. There is an analogous set of problems to blocking on the writes to the screen or to the remote end. If users have stopped output to their screens by typing the stop character, the write will block until they type the start character. In the meantime, the process cannot read from the keyboard to find out that the users want to flush the output.

Traditional UNIX systems have handled the multiplexing problem by using multiple processes that communicate through pipes or some other interprocess-communication facility, such as shared memory. This approach, however, can result in significant overhead as a result of context switching between the processes if the cost of processing input is small relative to the cost of a context switch. Furthermore, it is often more straightforward to implement applications of this sort in a single process. For these reasons, 4.3BSD provides two mechanisms that permit multiplexing I/O on descriptors: *nonblocking I/O* and *signal-driven I/O*. Operations on nonblocking descriptors complete immediately, partially complete an input or output operation and return a partial count, or return an error that indicates the operation could not be completed at all. Descriptors that have signaling enabled cause the associated process or process group to be notified when the I/O state of the descriptor changes.

There are four possible alternatives that avoid the blocking problem:

1. Set all the descriptors into nonblocking mode. The process can then try operations on each descriptor in turn to find out which descriptors are ready to do I/O. The problem with this approach is that the process must run continuously to discover whether there is any I/O to be done.
2. Enable all descriptors of interest to signal when I/O can be done. The process can then wait for a signal to discover when it is possible to do I/O. The drawback to this approach is that signals are expensive to catch. Hence, signal driven I/O is impractical for applications that do moderate to large amounts of I/O.
3. Have the system provide a method for asking which descriptors are capable of doing I/O. If none of the requested descriptors is ready, the system can put the process to sleep until a descriptor becomes ready. This approach avoids the problem of deadlock, since the process will be awakened whenever it is possible to do I/O, and will be told which descriptor is ready. It does have the drawback that the process must do two system calls per operation: one to poll for the descriptor that is ready to do I/O, and another to do the operation itself.

4. Have the process notify the system of all the descriptors that it is interested in reading, then do a blocking read on that set of descriptors. When the read returns, the process is notified on which descriptor the read completed. The benefit of this approach is that the process does a single system call to specify the set of descriptors, then loops doing only reads [Accetta *et al.*, 1986].

The third approach is used in 4.3BSD through the *select* system call. Although less efficient than the fourth approach, it is a more general interface. In addition to handling reading from multiple descriptors, it handles writes to multiple descriptors, notification of exceptional conditions, and timeout when no I/O is possible.

The *select* interface takes three masks of descriptors to be polled, corresponding to interest in reading, writing, and exceptional conditions. In addition, it takes a timeout value for returning from *select* if none of the requested descriptors becomes ready before a specified amount of time has elapsed. The *select* call returns the same three masks of descriptors after modifying them to show the descriptors that are able to do reading, to do writing, or to provide an exceptional condition. If none of the descriptors has become ready in the timeout interval, *select* returns indicating that no descriptors are ready for I/O.

Implementation of Select

The implementation of *select*, like that of many other kernel functions, is divided into a generic top layer and many device- or socket-specific bottom pieces.

At the top level, *select* decodes the request of the user and then arranges to call the appropriate lower-level select functions. The top level proceeds by taking several steps:

1. Fetch and validate the descriptor masks for read, write, and exceptional conditions. Validation requires checking that each requested descriptor is currently open by the process.
2. Set the *selecting* flag for the process.
3. For each descriptor in each mask, poll the device by calling its select routine. If the descriptor is not able to do the requested I/O operation, the select routine is responsible for recording that the process wants to do I/O. When I/O becomes possible for the descriptor, usually as a result of an interrupt from the underlying device, a notification must be issued for the selecting process.
4. If the process is no longer marked as selecting, mark the process as selecting and poll all the specified devices again. Because the selection process may take a long time, the kernel does not want to block out I/O during the time it takes to poll all the requested descriptors. Instead, the kernel arranges to detect the occurrence of I/O that may affect the status of the descriptors being polled. When such I/O occurs, the select-notification routine, *selwakeup()*, clears the *selecting* flag. If the top-level select code finds that the *selecting* flag for the

process has been cleared while it has been doing the polling, then the top level knows that the polling results may be invalid and must be repeated. The other condition that requires the polling to be repeated is caused by a *collision*. Collisions arise when multiple processes attempt to select on the same descriptor at the same time. Because the select routines have only enough space to record a single process identifier, they cannot track multiple processes that need to be awakened when I/O is possible. In such rare instances, all processes that are selecting must be awakened. If no descriptors are ready and the *select* specified a timeout, the kernel posts a timeout for the requested amount of time. The timeout function, *unselect()*, takes a pointer to the process-table entry for the process. The process goes to sleep, giving the address of the kernel global variable *selwait*. Normally, a descriptor will become ready and the process will be notified by *selwakeup()*. If none of the descriptors becomes ready before the timer expires, *unselect()* gets called and awakens the process. When the process is awakened, it repeats the polling process and returns the available descriptors. If it is awakened because of a timeout, the available descriptor list will be empty.

Each of the low-level polling routines in the terminal drivers and the network protocols follows roughly the same set of steps. A piece of the select routine for a terminal driver is shown in Fig. 6.4. The steps involved in a polling device are as follows:

1. The socket or device select entry is called with flag of FREAD, FWRITE, or 0 (exceptional condition). The example in Fig. 6.4 shows the FREAD case; the others cases are similar.
2. The poll returns success if the requested operation is possible. In Fig. 6.4, it is possible to read a character if the number of unread characters is greater than zero. In addition, if the carrier has dropped, it is possible to get a read error: A

Figure 6.4 Select for reading in a terminal driver.

```
struct tty *tp;

case FREAD:
    if (nread > 0 || (tp->t_state & TS_CARR_ON) == 0)
        return (1);
    if (tp->t_rsel &&
        tp->t_rsel->p_wchan == (caddr_t)&selwait)
        tp->t_state |= TS_RCOLL;
    else
        tp->t_rsel = u.u_procp;
    return (0);
```


return from select does not necessarily mean that there are data to read; rather, it means that a read will not block.

3. If the requested operation is not possible, the process identifier is recorded with the socket or device for later notification. In Fig. 6.4, the second **if** statement is false and the process identifier (a pointer to the process table entry for the process) is saved in *t_rsel*.
4. If multiple processes are selecting on the same socket or device, a collision is recorded for the socket or device because the structure has only enough space for a single process identifier. In Fig. 6.4, a collision occurs when the second **if** statement is true.

There is a *tty* structure for each terminal line (or pseudoterminal) on the machine. Normally, only one process at a time is selecting to read from the terminal. The second **if** statement in Fig. 6.4 detects the case when more than one process is selecting at the same time. The check for a collision is as follows. The first part of the conjunctive checks to see whether any process identifier is recorded already. If there is none, then there is no collision. If there is a process identifier recorded, it may remain from an earlier call on select by a process that is no longer selecting because one of its other descriptors became ready. If that process is still selecting, it will be sleeping on *selwait* (when sleeping, the address of the sleep event is stored in *p_wchan*). If it is sleeping on some other event, its *p_wchan* will have a value different from that of *selwait*. If it is running, its *p_wchan* will be zero. Usually, *t_rsel* is set to the current process, but, since the current process is running (in the select code), the value for its *p_wchan* will be zero and no collision will occur.

Selecting processes must be notified when I/O becomes possible. The steps involved in detecting and awakening a process are as follows:

1. The device or socket detects a change in status. Status changes normally occur because of an interrupt (for example, a character coming in from a keyboard or a packet coming in from the network).
2. *Selwakeup()* is called with the process identifier and a flag showing whether a collision occurred.
3. If the process is sleeping on *selwait*, it is made runnable (or is marked ready if it is stopped). If the process is sleeping on some event other than *selwait*, it is not made runnable. A spurious call to *selwakeup()* can occur when the process returns from *select* to begin processing one descriptor and then another descriptor on which it had been selecting also becomes ready.
4. If the process has its *selecting* flag set, the flag is cleared so that it will know that its polling results are invalid and must be recomputed.
5. If a collision has occurred, all sleepers on *selwait* are awakened to rescan to see whether one of their descriptors became ready. Awakening all selecting

processes is necessary because the low-level select routine could not record all the processes that needed to be awakened. Hence, it has to wake up all processes that could possibly have been interested. Empirically, collisions occur very infrequently. If they were a frequent occurrence, it would be worthwhile to design a more complex data structure to hold multiple process identifiers.

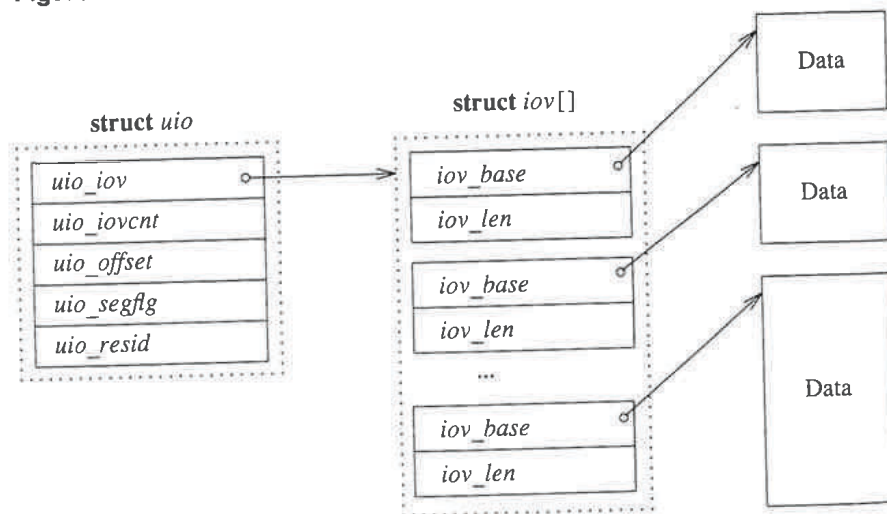
Moving Data Inside the Kernel

Within the kernel, I/O data are described by an array of vectors. Each vector has a base address and a length, and is called an *I/O vector* or *iovec*. The I/O vectors are identical to the I/O vectors used by the *readv* and *writv* system calls.

The kernel maintains another structure that holds additional information about the I/O operation called a *uio* structure. A sample *uio* structure is shown in Fig. 6.5; it contains

- A pointer to the *iovec* array
- The size of the *iovec* array
- The file offset at which the operation should start
- A flag showing whether the source and destination are both within the kernel, or whether the source and destination are split between the user and the kernel
- The sum of the lengths of the I/O vectors

Figure 6.5 A *uio* structure.



All I/O within the kernel is described with *iovec* and *uio* structures. System calls such as *read* and *write* that are not passed an *iovec* create a *uio* to describe their arguments; this *uio* structure is passed to the lower levels of the kernel to specify the parameters of an I/O operation. Eventually, the *uio* structure reaches the part of the kernel responsible for moving the data to or from the process address space—the filesystem, the network, or a device driver. In general, these parts of the kernel do not interpret *uio* structures directly. Instead, they arrange a kernel buffer to hold the data, then use *uiomove()* to copy the data to or from the buffer(s) described by the *uio* structure. The *uiomove()* routine is called with a pointer to a kernel data area, a data count, and a *uio* structure. As it moves data, it updates the counters and pointers of the *iovec* and *uio* structures by a corresponding amount. If the kernel buffer is not as large as the areas described by the *uio* structure, the *uio* structure will point to the part of the process address space just beyond the last location completed. Thus, while servicing a request, the kernel may call *uiomove()* multiple times, each time giving a pointer to a new kernel buffer for the next block of data.

Character device drivers that do not copy data from the process generally do not interpret the *uio* structure themselves either. Instead, there is one low-level kernel routine that arranges a direct transfer to or from the address space of the process. Here, a separate I/O operation is done for each *iovec* element, calling back to the driver with one piece at a time.

Traditional UNIX systems use global variables in the user area to describe I/O. This approach has several problems. The lower levels of the kernel are not reentrant, since there is exactly one context to describe an I/O operation. The system cannot do scatter-gather I/O, since there is only a single base and size variable per process. Finally, the bottom half of the kernel cannot do I/O, since it does not have a user area.

The one part of the 4.2BSD kernel that does not use *uio* structures is the block device drivers. The decision not to change these interfaces to use *uio* structures was largely pragmatic. The developers would have had to change many drivers. The existing buffer interface was already decoupled from the user structure; hence, the interface was already reentrant and could be used by the bottom half of the kernel. The only gain was to allow scatter-gather I/O. The kernel does not need scatter-gather operations on block devices, however, and user operations on block devices are done through the buffer cache.

Exercises

- 6.1 Where are the read and write attributes of an open file descriptor stored?
- 6.2 Why is the close-on-exec bit located in the per-process descriptor table instead of in the system file table?
- 6.3 Why are the file-table entries reference counted?

- 6.4 What shortcomings of lock files are addressed by the 4.3BSD descriptor-locking facilities?
- 6.5 What problems are raised by mandatory locks?
- 6.6 Why is the implementation of *select* split between the descriptor-management code and the lower-level routines?
- 6.7 Describe how the *process selecting flag* is used in the implementation of *select*.
- 6.8 The *update* program is usually started shortly after the system is booted. Once every 30 seconds, it does a *sync* system call. What problems could arise if this program were not run?
- 6.9 The special device */dev/kmem* provides access to the kernel's virtual address space. Would you expect it to be a character or a block device? Explain your answer.
- 6.10 Many tape drives provide a block-device interface. Is it possible to support a filesystem on a such a tape drive?
- *6.11 True asynchronous I/O is not supported in 4.3BSD. What problems arise with providing asynchronous I/O in the existing read-write interface?
- **6.12 Describe an alternative scheme to the per-process descriptor table that does not impose an upper bound on the number of open files that a process may have. What problems are created for your scheme by the requirement that open file descriptors be inherited across a *fork*?

References

- Accetta *et al.*, 1986.
M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, & M. Young, "Mach: A New Kernel Foundation for UNIX Development," Technical Report, Carnegie Mellon University, Pittsburgh, PA (August 1986).
- Bass, 1981.
J. Bass, *Implementation Description for File Locking*, Onyx Systems Inc., 73 E. Trimble Road, San Jose, CA (January 1981).
- Kenah & Bate, 1984.
L. J. Kenah & S. F. Bate, *VAX/VMS Internals and Data Structures*, Digital Press, Bedford, MA (1984).
- Peterson, 1983.
G. Peterson, "Concurrent Reading while Writing," *ACM Transactions on Programming Languages and Systems* 5(1), pp. 46-55 (January 1983).
- Shaw, 1974.
A. C. Shaw, *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ (1974).

CHAPTER 7

The Filesystem

7.1 Structure and Overview

4.3BSD organizes data in filesystems that reside on mass-storage media such as disk drives. Each disk drive is divided into one or more subdivisions, or partitions. Each such partition may contain only one filesystem, and a filesystem never spans multiple partitions.

Within the filesystem are files, most of which contain ordinary data. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. This hierarchy of directories and files is organized into a tree structure; Fig. 7.1 shows a small filesystem tree.

Directories

Directories are allocated in units called *chunks*; Fig. 7.2 shows a typical directory chunk. The size of a chunk is chosen such that each allocation can be transferred to disk in a single operation; the ability to change a directory in a single operation allows directory updates to be atomic. Chunks are broken up into variable-length directory entries to allow filenames to be of nearly arbitrary length. No directory entry is allowed to span multiple chunks. The first three fields of a directory entry are fixed length and contain

1. A pointer to an on-disk data structure describing the file; the contents of this on-disk data structure, which is called an *inode*, are described in Section 7.2
2. The size of the entry
3. The length of the filename contained in the entry

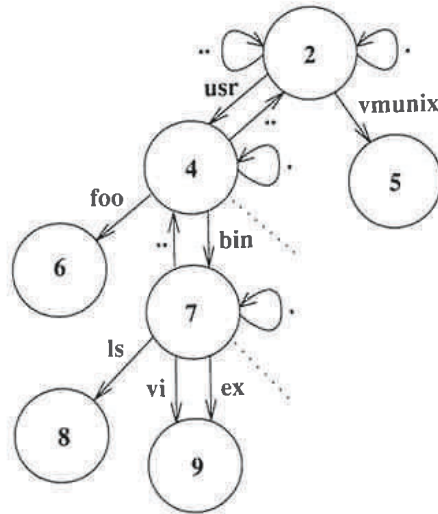
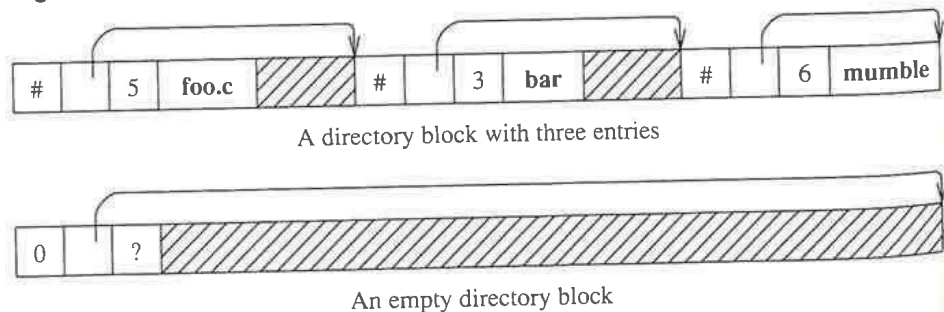


Figure 7.1 A small filesystem tree.

The remainder of an entry is variable length and contains a null-terminated filename, padded to a 4-byte boundary. The maximum length of a filename in a directory is 255 characters.

Available space in a directory is recorded by having one or more entries accumulate the free space in their size fields. This results in directory entries that are larger than required to hold the entry name plus fixed-length fields. Space allocated to a directory should always be completely accounted for by the total of the sizes of its entries. When an entry is deleted from a directory, the system coalesces its space into the previous entry in the same directory chunk by increasing the size of the previous entry by the size of the deleted entry. If the first entry of a directory chunk is free, then the pointer to the entry's inode is set to zero to show that it is unallocated.

Figure 7.2 Format of directory entries.



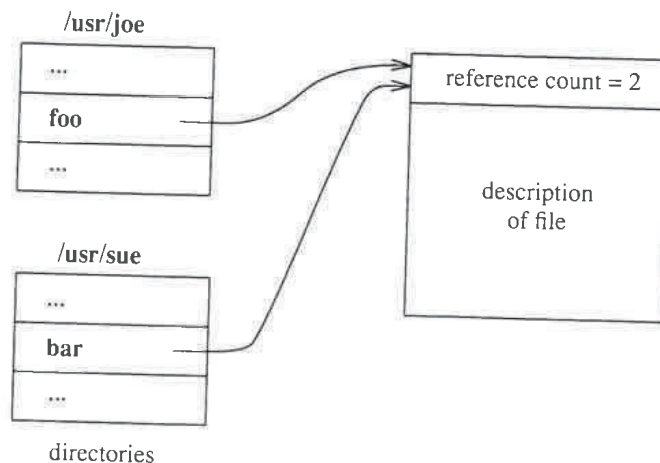
The new directory structure was introduced in 4.2BSD. Other UNIX systems use 14-character filenames. Directories consist of arrays of 16-byte entries: The first two bytes are the inode number; the remaining 14 bytes are the filename. Programs scan directories by reading the latter as ordinary files with *open*, *read*, *lseek*, and *close*. 4.2BSD introduced a new library interface that could be used to access either the new or the old directory structure. The *opendir()* function returns a structure pointer that is used by *readdir()* to obtain the next directory entry, by *rewinddir()* to reset the read position to the beginning, and by *closedir()* to close the directory. In addition, there is *seekdir()*, which returns to a position previously obtained with *telldir()*.

Another benefit of using directory-access routines is that they can be emulated over a network, or even for a non-UNIX filesystem such as MS-DOS.

Links

Every file has an associated inode that contains information describing access permissions, ownership of the file, timestamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. Each file has a single inode, but may have multiple directory entries in the same filesystem to reference it (i.e., multiple names). Each directory entry creates a *hard link* of a filename to the inode that describes its contents. The link concept is fundamental; inodes do not reside in directories, but exist separately and are referenced by links. When all the links to an inode are removed, the inode is deallocated. If one link to a file is removed and the filename is recreated with new contents, the other links will continue to point to the old inode. Figure 7.3 shows two different directory entries, *foo* and *bar*, that reference the same file; thus, the inode for the file shows a reference count of two.

Figure 7.3 Hard links to a file.

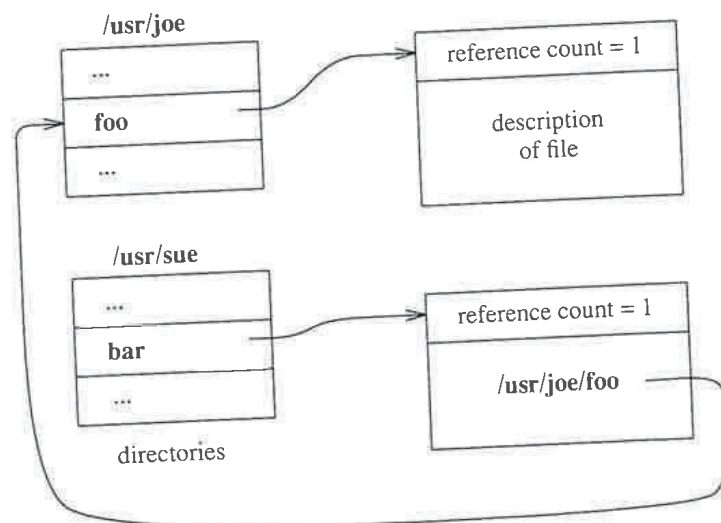


The system also supports a *symbolic link*, or *soft link*. A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link are prepended to the rest of the pathname, and this name is interpreted to yield the resulting pathname. If a symbolic link contains an absolute pathname, that absolute pathname is used; otherwise, the contents of the symbolic link are evaluated relative to the location of the link in the file hierarchy (not according to the current working directory of the calling process).

An example symbolic link is shown in Fig. 7.4. Here there is a hard link, **foo**, that points to the file. The other reference, **bar**, points to a different inode whose contents is a pathname of the referenced file. When a process opens **bar**, the system interprets the contents of the symbolic link as a pathname to find the file the link references. Symbolic links are treated like data files by the system, rather than as part of the filesystem structure; thus, they can point at directories or files on other filesystems. If a filename is removed and replaced, any symbolic links that point to it will access the new file. Finally, if the filename is not replaced, the symbolic link will point at nothing and any attempt to access it will cause an error to be returned.

When *open* is applied to a symbolic link, it returns a file descriptor for the file pointed to, not for the link itself. Otherwise, it would be necessary to use some sort of indirection to access the file pointed to—and that file, not the link, is what is usually wanted. For the same reason, most other system calls that take pathname arguments also follow symbolic links. The *stat* system call is an exception to this rule, because it is sometimes useful to be able to detect a symbolic link when traversing a filesystem or when making an archive tape. So another variant

Figure 7.4 Symbolic link to a file.



of *stat*, *lstat*, is available to get the status of a symbolic link instead of the object at which that link points.

A symbolic link has several advantages over a hard link. Specifically, a symbolic link can refer to a directory or to a file on a different filesystem. A hard link is not permitted to refer to a directory, to prevent loops in the filesystem hierarchy. The implementation of hard links prevents them from referring to files on a different filesystem.

There are some interesting implications of symbolic links. Consider a process that has current working directory */usr/sue* and does *cd src*, where *src* is a symbolic link to directory */usr/src*. If the process then does a *cd ..*, then it will end up in */usr* instead of in */usr/sue*, as it would have done if *src* was a normal directory instead of a symbolic link. The kernel could be changed to keep track of the symbolic links that a process had traversed, and to interpret *..* differently if the directory had been reached through a symbolic link. There are two problems with this implementation. First, the kernel would have to maintain a potentially unbounded amount of information. Second, no program could depend on being able to use *..*, since it could not be sure how the name would be interpreted.

Since symbolic links may cause loops in the filesystem, the kernel checks for a large number (eight) of symbolic link traversals in a single pathname translation. If the limit is reached, the kernel produces an error (ELOOP).

Quotas

Resource sharing traditionally has been a design goal for the UNIX system. In some implementations, any single user can allocate all the available space in the filesystem. In certain environments, this is unacceptable. Consequently, 4.3BSD includes a quota mechanism to restrict the amount of filesystem resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. Quotas can be set separately for each user on each filesystem.

Resources are given both a hard and a soft limit. When a process exceeds a soft limit, a warning is printed on the user's terminal; the offending process is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but may use more resources while they are working actively. To encourage this behavior, the system warns users when they log in if they are over any of their soft limits. If a user fails to correct the problem for too many login sessions, the soft limit starts being enforced as the hard limit. These quotas are derived from a larger resource-limit package that was developed at the University of Melbourne in Australia by Elz [Elz, 1984].

7.2 Overview of the Internal Filesystem

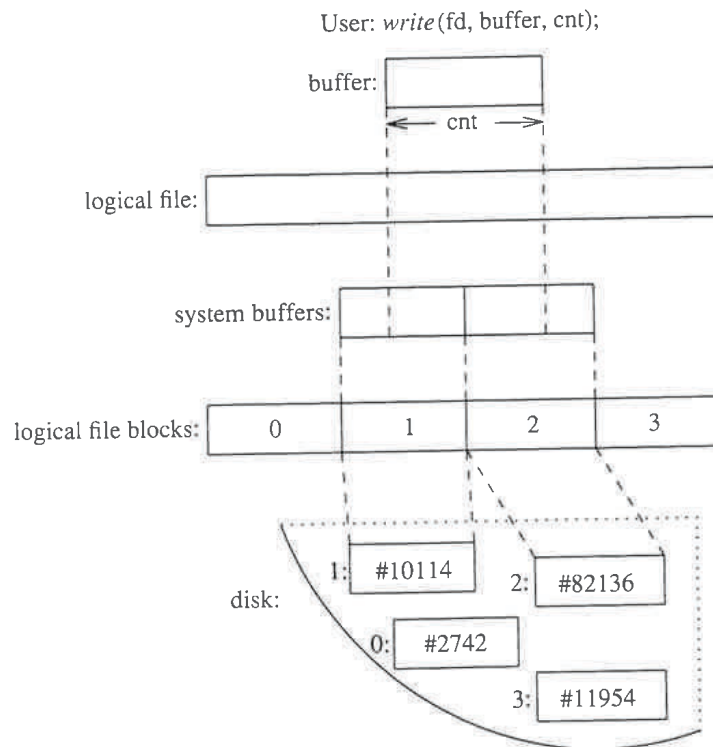
The filesystem implementation converts from the user abstraction of a file as an array of bytes to the structure imposed by the underlying physical medium. Consider a typical medium of a magnetic disk with fixed-sized sectoring. Although the user may wish to write a single byte to a file, the disk can read and write only

in multiples of sectors. Here, the system must arrange to read in the sector containing the byte to be modified, to replace the affected byte, and to write the sector back to the disk. This operation of converting random access to an array of bytes to reads and writes of disk sectors is known as *block I/O*.

First, the system breaks the user's request into a set of operations to be done on each *logical block* of the file. Logical blocks describe block-sized pieces of a file. The system calculates the logical blocks by dividing the array of bytes into filesystem-sized pieces. Thus, if a filesystem's block size is 8192 bytes, then logical block 0 would contain bytes 0 to 8191, logical block 1 would contain bytes 8192 to 16,383, and so on.

Figure 7.5 shows the flow of information and work required to access the filesystem on the disk. The abstraction shown to the user is an array of bytes. These bytes are collectively described by a file descriptor that refers to some location in the array. The user can request a write operation on the file by presenting the system with a pointer to a buffer, along with a request for some number of bytes to be written. Figure 7.5 shows that the requested data need not be aligned with the beginning or end of a disk sector. Further, the size of the request is not constrained to a single disk sector. In the example shown, the user has requested data to be written to parts of logical blocks 1 and 2.

Figure 7.5 The block input/output system.



The data in each logical block are stored in a *physical block* on the disk. A physical block is the location on the disk to which the system maps a logical block. A physical disk block is constructed from one or more contiguous sectors. For a disk with 512-byte sectors, an 8192-byte filesystem block would be built up from 16 contiguous sectors. Although the contents of a logical block are contiguous on disk, the logical blocks of the file need not be laid out contiguously. The next section describes the data structure used by the system to convert from logical blocks to physical blocks.

Returning to our example in Fig. 7.5, we have now figured out which logical blocks are to be updated. Since the disk can transfer data only in multiples of sectors, the filesystem must first arrange to read in the data for the part of the block that is to be left unchanged. The system must arrange an intermediate staging area for the transfer. This staging is done through one or more system buffers.

In our example, the user wishes to modify data in logical blocks 1 and 2. The operation iterates over five steps:

1. Allocate a buffer
2. Determine the location of the corresponding physical block on the disk
3. Request the disk controller to read the contents of the physical block into the system buffer and wait for the transfer to complete
4. Do a memory-to-memory copy from the beginning of the user's I/O buffer to the appropriate portion of the system buffer
5. Write the block to the disk

If the user's request is incomplete, the process is repeated with the next logical block of the file. In our example, the system fetches logical block 2 of the file and is able to complete the user's request. Had an entire block been written, the system could have skipped step 3 and have simply written the data to the disk without first reading in the old contents. This incremental filling of the write request is transparent to the user's process because that process is blocked from running during the whole filling procedure. The filling is transparent to other processes; since the inode is locked during the whole process, any attempted access by any other process will be blocked until the initial write has completed.

Allocating and Finding the Blocks on the Disk

The filesystem also handles the allocation of new blocks to files as the latter grow. Simple filesystem implementations, such as those used by early microcomputer systems, allocate files contiguously, one after the next, until the files reach the end of the disk. As files are removed, holes occur. To reuse the freed space, the system must compact the disk to move all the free space to the end. Files can be created only one at a time; for the size of a file other than the last one on the disk to be increased, the file must be copied to the end, then expanded.

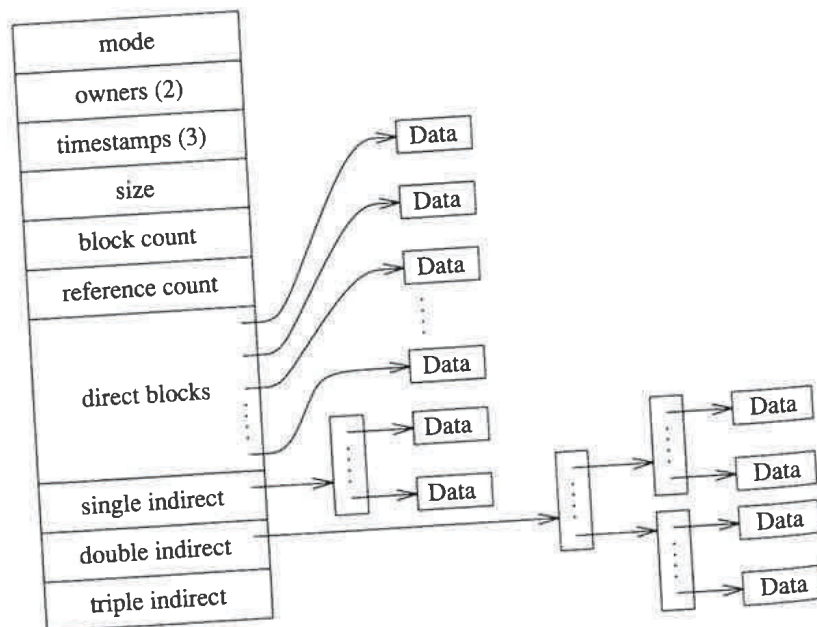
To allow both multiple file allocation and random access, most UNIX systems use the concept of an *index node*, or *inode*. Figure 7.6 shows that the inode contains an array of pointers to the blocks in the file. The system can convert from a logical block number to a physical sector number by indexing into the array using the logical block number. A null array entry indicates that no block has been allocated, and will cause a block of zeros to be returned on a read. On a write of such an entry, a new block is allocated, the array entry is updated with the new block number, and the data are written to the disk.

Inodes are statically allocated and most files are small, so the array of pointers must be small for efficient use of space. The first 12 array entries are allocated in the inode itself. For typical filesystems, this allows the first 48 or 96 Kbyte of data to be located directly using a simple indexed lookup.

For somewhat larger files, Fig. 7.6 shows that the inode contains a *single indirect pointer* that points to a *single indirect block* of pointers to data blocks. To find the one-hundredth logical block of a file, the system first fetches the block identified by the indirect pointer, then indexes into the eighty-eighth block (100 - 12 direct pointers), and fetches that data block.

For files that are bigger than a few Mbytes, the single indirect block is eventually exhausted; these files must resort to using a *double indirect block*, which is a pointer to a block of pointers to pointers to data blocks. Finally, the system leaves provision for a *triple indirect block*, although such a block is not used in existing implementations.

Figure 7.6 The structure of an inode.



Although single and double indirect blocks appear to double or triple the number of disk accesses required to get a block of data, the overhead of the transfer is typically much lower. In Section 7.5, we shall discuss the management of the filesystem cache that holds recently used disk blocks. The first time that a block of indirect pointers is needed, it is brought into the filesystem cache. Further accesses to the indirect pointers find the block already resident in memory; thus, they require only a single disk access to get the data.

In addition to the logical-to-physical block mapping, the inode contains the other information about the contents of the file. This information includes

- The type and access mode for the file
- The file's owner
- The group-access identifier
- The number of references to the file
- The time the file was last read and written
- The time that the inode was last updated by the system
- The size of the file in bytes
- The number of physical blocks used by the file (including blocks used to hold indirect pointers)

Notably missing in the inode is the filename. Filenames are maintained in directories rather than in inodes because a file may have many names, or links, and the name of a file may be large (up to 255 bytes in length).

To create a new name for a file, the system increments the count of the number of names referring to that inode. Then, the new name is entered in a directory, along with the number of the inode. Conversely, when a name is deleted, the entry is deleted from a directory and the name count for the inode is then decremented. When the name count reaches zero, the system deallocates the inode by putting all the inode's blocks back on a list of free blocks and putting the inode back on a list of unused inodes.

7.3 Internal Structure and Redesign

A traditional UNIX filesystem, such as that of System V, is described by its superblock, which contains the basic parameters of the filesystem. These parameters include the number of data blocks in the filesystem, a count of the maximum number of files, and a pointer to the *free list*, which is a list of all the free blocks in the filesystem.

A 150-Mbyte traditional UNIX filesystem consists of 4 Mbyte of inodes followed by 146 Mbyte of data. That organization segregates the inode information from the data; thus, accessing a file normally incurs a long seek from the file's

inode to its data. Files in a single directory typically are not allocated consecutive slots in the 4 Mbyte of inodes, causing many nonconsecutive disk blocks to be read when many inodes in a single directory are accessed.

The allocation of data blocks to files also is suboptimal. The traditional filesystem implementation uses a 512-byte physical block size. But the next sequential data block often is not on the same cylinder, so seeks between 512-byte data transfers are required frequently. This combination of small block size and scattered placement severely limits filesystem throughput.

The first work on the UNIX filesystem at Berkeley attempted to improve both the reliability and the throughput of the filesystem. The developers improved reliability by staging modifications to critical filesystem information so that the modifications could be either completed or repaired cleanly by a program after a crash [McKusick & Kowalski, 1986]. They improved the performance of the filesystem by a factor of more than two between 3BSD and 4.0BSD by doubling the block size. This doubling caused each disk transfer to access twice as many data and eliminated the need for indirect blocks for many files. We shall refer to the filesystem with these changes in the rest of this section as the *old filesystem*.

The performance improvement in the old filesystem gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old filesystem was still using only about 4 percent of the maximum disk throughput. The main problem was that the order of blocks on the free list quickly became scrambled as files were created and removed. Eventually, the free-list order became entirely random, causing files to have their blocks allocated randomly over the disk. This forced a seek before every block access. Although the old filesystem provided transfer rates of up to 175 Kbyte per second when it was first created, the scrambling of the free list caused this rate to deteriorate to an average of 30 Kbyte per second after a few weeks of moderate use. There was no way of restoring the performance of an old filesystem except to recreate the system.

New Filesystem Organization

In the 4.3BSD filesystem organization (as in the old filesystem organization), each disk drive contains one or more filesystems. A 4.3BSD filesystem is described by its *superblock*, located at the beginning of the filesystem's disk partition. Because the superblock contains critical data, it is replicated to protect against catastrophic loss. This replication is done when the filesystem is created; since the superblock data do not change, the copies need not be referenced unless a disk failure causes the default superblock to be corrupted.

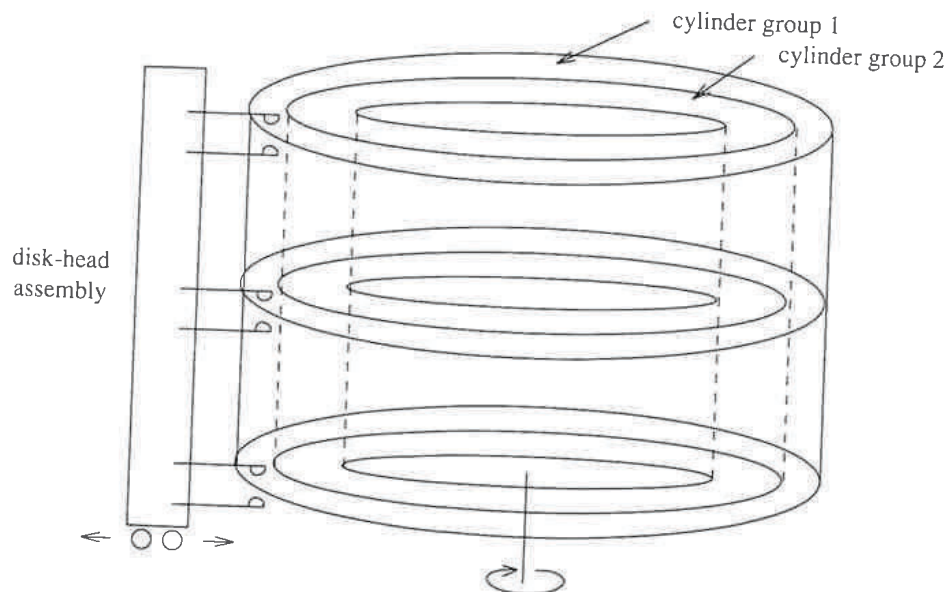
To ensure that files as large as 2^{32} bytes may be created with only two levels of indirection, the minimum size of a filesystem block is 4096 bytes. The block size can be any power of two greater than or equal to 4096. The block size is recorded in the filesystem's superblock, so it is possible for filesystems with different block sizes to be accessible simultaneously on the same system. The block size must be selected at the time that the filesystem is created; it cannot be changed subsequently without rebuilding of the filesystem.

The new filesystem organization divides a disk partition into one or more areas, each of which is called a *cylinder group*. Figure 7.7 shows a set of cylinder groups, each comprising one or more consecutive cylinders on a disk. Each cylinder group contains bookkeeping information that includes a redundant copy of the superblock, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. The bit map of available blocks in the cylinder group replaces the traditional filesystem's free list. For each cylinder group, a static number of inodes is allocated at filesystem creation time. The default policy is to allocate one inode for each 2048 bytes of space in the cylinder group, with the expectation that this amount will be far more than will ever be needed.

The motivation for cylinder groups is to create clusters of inodes that are spread over the disk, instead of them all being located at the beginning of the disk. The filesystem attempts to allocate file blocks close to the inodes that describe them to avoid long seeks between getting the inode and getting its associated data. Also, when the inodes are spread out, there is less chance of losing all of them in a single disk failure.

All the bookkeeping information could be placed at the beginning of each cylinder group. If this approach were used, however, all the redundant information would be on the same platter of a disk. A single hardware failure could then destroy all copies of the superblock. Thus, the bookkeeping information begins at a varying offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track farther from the

Figure 7.7 Layout of cylinder groups.



beginning than is the preceding cylinder group. In this way, the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without all copies of the superblock also being lost. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder-group information is used for data blocks.

Optimizing Storage Utilization

Data are laid out so that larger blocks can be transferred in a single disk operation, greatly increasing filesystem throughput. A file in the new filesystem might be composed of 4096-byte data blocks, as compared to the 1024-byte blocks of the old filesystem: Disk accesses would thus transfer up to four times as much information per disk transaction. In large files, several blocks may be allocated from the same cylinder, so that even larger data transfers are possible before a seek is required.

The main problem with larger blocks is that most UNIX filesystems contain primarily small files. A uniformly large block size wastes space. Table 7.1 shows the effect of filesystem block size on the amount of wasted space in the filesystem. The files measured to obtain these figures reside on a time-sharing system at Berkeley that has roughly 1.2 gigabytes of on-line storage. The measurements are based on the active user filesystems containing about 920 Mbyte of formatted space. The space wasted is calculated to be the percentage of disk space not containing user data. As the block size increases, the waste rises quickly to an intolerable 45.6 percent waste with 4096-byte filesystem blocks.

For large blocks to be used without significant waste, small files must be stored more efficiently. The developers obtained this efficiency by allowing the division of a single filesystem block into one or more *fragments*. The fragment size is specified at the time that the filesystem is created; each filesystem block optionally can be broken into two, four, or eight fragments, each of which is

Table 7.1 Amount of space wasted as a function of block size.

Space used (Mbyte)	Percent waste	Organization
775.2	0.0	Data only, no separation between files
807.8	4.2	Data only, each file starts on 512-byte boundary
828.7	6.9	Data + inodes, 512-byte block UNIX filesystem
866.5	11.8	Data + inodes, 1024-byte block UNIX filesystem
948.5	22.4	Data + inodes, 2048-byte block UNIX filesystem
1128.3	45.6	Data + inodes, 4096-byte block UNIX filesystem

bits in map	----	--11	11--	1111
fragment numbers	0-3	4-7	8-11	12-15
block numbers	0	1	2	3

Figure 7.8 Example of the layout of blocks and fragments in a 4096/1024 filesystem.

addressable. The lower bound on the fragment size is constrained by the disk-sector size, which is typically 512 bytes. The block map associated with each cylinder group records the space available in a cylinder group in fragments; to determine whether a block is available, the system examines aligned fragments. Figure 7.8 shows a piece of a block map from a filesystem with 4096-byte blocks and 1024-byte fragments, hereinafter referred to as a 4096/1024 filesystem. Each bit in the map records the status of a fragment; an “-” shows that the fragment is in use, whereas a “1” shows that the fragment is available for allocation. In this example, fragments 0 through 5, 10, and 11 are in use, whereas fragments 6 through 9 and 12 through 15 are free. Fragments of adjacent blocks cannot be used as a full block, even if they are large enough. In this example, fragments 6 through 9 cannot be allocated as a full block; only fragments 12 through 15 can be coalesced into a full block.

On a filesystem with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096-byte blocks of data, possibly plus a single fragmented block. If a block must be fragmented to obtain space for a small number of data, the remaining fragments of the block are made available for allocation to other files. As an example, consider an 11000-byte file stored on a 4096/1024 filesystem. This file would use two full-sized blocks and one three-fragment portion of another block. If no block with three aligned fragments were available at the time the file was created, a full-sized block would be split, yielding the necessary fragments and a single unused fragment. This remaining fragment could be allocated to another file as needed.

Each time a process does a *write* system call, the system checks to see whether the size of the file has increased. A process may overwrite data in the middle of an existing file, in which case space would have been allocated already. If the file needs to be expanded, the request is rounded up to the next fragment size and only that much space is allocated. Many small write requests may expand the file one fragment at a time. The problem with expanding a file one fragment at a time is that data may be copied many times as a fragmented block expands to a full block. Fragment reallocation can be minimized if the user process writes a full block at a time, except for a partial block at the end of the file. Since filesystems with different block sizes may reside on the same system, the filesystem interface provides application programs with the optimal size for a read or write. This facility is used by the standard I/O library that many application programs use, and by certain system utilities, such as archivers and loaders, that do their own I/O management.

The amount of wasted space in the 4096/1024 new filesystem organization is empirically observed to be about the same as that in the old 1024-byte filesystem organization. A filesystem with 4096-byte blocks and 512-byte fragments has about the same amount of wasted space as the 512-byte block UNIX filesystem. The new filesystem uses less space than the 512-byte or 1024-byte filesystems for indexing information for large files, and the same amount of space for small files. These savings are offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk-space utilization when a new filesystem's fragment size is equal to an old filesystem's block size.

For the layout policies (described at the end of this section) to be effective, a filesystem cannot be kept completely full. A parameter, termed the *free-space reserve*, gives the minimum percentage of filesystem blocks that should be kept free. If the number of free blocks drops below this level, only the superuser is allowed to allocate blocks. This parameter may be changed at any time. When the number of free blocks approaches zero, the filesystem throughput tends to be cut in half, because the filesystem is unable to localize blocks in a file. If a filesystem's throughput drops because of overfilling, it may be restored by removal of files until the amount of free space once again reaches the minimum acceptable level. Users can restore access rates for files created during periods of little free space by moving the file's data when enough space is available. We must add the free-space reserve to the percentage of waste when comparing the organizations given in Table 7.1. Thus, the percentage of waste in an old 1024-byte UNIX filesystem is roughly comparable to a new 4096/512 filesystem with the free-space reserve set at 5 percent. (Compare 11.8 percent wasted with the old filesystem to 6.9 percent waste + 5 percent reserved space in the new filesystem.)

Filesystem Parameterization

Except for the initial creation of the free list, the old filesystem ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass-storage device, or the hardware that interacts with the filesystem. A goal of the new filesystem is to parameterize the processor capabilities and mass-storage characteristics so that blocks can be allocated in an optimum configuration-dependent way. Important parameters include the speed of the processor, the hardware support for mass-storage transfers, and the characteristics of the mass-storage devices. Disk technology is constantly improving, and a given installation can have several different disk technologies running on a single processor. Each filesystem is parameterized so that it can be adapted to the characteristics of the disk on which it is located.

For mass-storage devices such as disks, the new filesystem tries to allocate a file's new blocks on the same cylinder and rotationally well positioned. The distance between *rotationally optimal* blocks varies greatly; optimal blocks can be consecutive or rotationally delayed, depending on system characteristics. For disks attached to a dedicated I/O processor, two consecutive disk blocks often can be accessed without time lost because of an intervening disk revolution. For processors without a dedicated I/O processor, the main processor must field an

interrupt and prepare for a new disk transfer. The expected time to service this interrupt and to schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to service an interrupt and to schedule a new disk transfer. Given a block allocated to a file, the allocation routines calculate the number of blocks to skip over such that the next block in the file will come into position under the disk head in the expected amount of time that it takes to start a new disk-transfer operation. For sequential access to large numbers of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time. If a filesystem is parameterized to lay out blocks with a rotational separation of 2 milliseconds, and the disk is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the target machine is known, the filesystem can be parameterized for that machine even though it is initially created on a different processor. Even if the move is not known in advance, the rotational-layout delay can be reconfigured after the disk is moved, so that all further allocation is done based on the characteristics of the new device.

Layout Policies

The filesystem layout policies are divided into two distinct parts. At the top level are global policies that use summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there is insufficient space left in the current cylinder group to do reasonable layouts. Below the global-policy routines are the local-allocation routines. These routines use a locally optimal scheme to lay out data blocks. The original intention was to bring out these decisions to user level, so that they could be ignored or replaced by user processes. They are thus definitely policies, not simple mechanisms.

Two methods for improving filesystem performance are to increase the locality of reference to minimize seek latency [Trivedi, 1980], and to improve the layout of data to make larger transfers possible [Nevalainen & Vesterinen, 1977]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must instead try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space, forcing the data to be scattered to nonlocal cylinder groups. Taken to an extreme, total localization can

result in a single huge cluster of data resembling the old filesystem. The global policies try to balance the two conflicting goals of localizing data that are concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes of files in the same directory frequently are accessed together. For example, the list-directory command, `ls`, may access the inode for each file in a directory. The inode layout policy tries to place all the inodes of files in a directory in the same cylinder group. To ensure that files are distributed throughout the filesystem, the system uses a different policy to allocate directory inodes. New directories are placed in cylinder groups with a greater than average number of free inodes, and with the smallest number of directories. The intent of this policy is to allow inode clustering to succeed most of the time. The allocation of inodes within a cylinder group is done using a next-free strategy. Although this method allocates the inodes randomly within a cylinder group, all the inodes for a particular cylinder group can be accessed with eight to 16 disk transfers. (At most 16 disk transfers are required because a cylinder group may have no more than 2048 inodes.) This allocation strategy puts a small and constant upper bound on the number of disk transfers required to access the inodes for all the files in a directory. In contrast, the old filesystem typically requires one disk transfer to fetch the inode for each file in a directory.

The other major resource is data blocks. Because data blocks for a file typically are accessed together, the policy routines try to place data blocks for a file in the same cylinder group, preferably at rotationally optimal positions in the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files quickly use up available space, forcing a spillover to other areas. Further, using all the space causes future allocations for any file in the cylinder group also to spill to other areas. Ideally, none of the cylinder groups should ever become completely full. The heuristic solution chosen is to redirect block allocation to a different cylinder group after every 1 Mbyte of allocation. The spillover points are intended to force block allocation to be redirected when a file has used about 25 percent of the data blocks in a cylinder group. In day-to-day use, the heuristics appear to work well in minimizing the number of completely filled cylinder groups.

The newly chosen cylinder group for block allocation is selected from the next cylinder group that has a greater-than-average number of free blocks left. Although big files tend to be spread out over the disk, 1 Mbyte of data is typically accessible before a long seek must be performed, and the cost of one long seek per Mbyte is small. It is also important to force large files to move before using up all the available blocks in a cylinder group, because the last few blocks will be scattered around the cylinder group and thus will require many small seeks to access.

The global-policy routines call local-allocation routines with requests for specific blocks. The local-allocation routines will always allocate the requested block if it is free; otherwise, they will allocate a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation

routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the global layout policy uses heuristics that employ only partial information.

If a requested block is not available, the local allocator uses a four-level allocation strategy:

1. Use the next available block rotationally closest to the requested block on the same cylinder. It is assumed that head-switching time is zero. On disk controllers where this is not the case, it may be possible to incorporate the time required to switch between disk platters when the rotational layout tables are constructed. This, however, has not yet been attempted.
2. If no blocks are available on the same cylinder, use a block within the same cylinder group.
3. If the cylinder group is full, quadratically hash the cylinder group number to choose another cylinder group to look for a free block.
4. Finally, if the hash fails, apply an exhaustive search to all cylinder groups.

Quadratic hash is used because of its speed in finding unused slots in nearly full hash tables [Knuth, 1975]. Filesystems that are parameterized to maintain at least 10 percent free space rarely use this strategy. Filesystems used without free space typically have so few free blocks available that almost any allocation is random; the most important characteristic of the strategy used under such conditions is that it be fast.

7.4 Filesystem Data Structures

We are now ready to look at the filesystem data structures in more detail. The small filesystem introduced in Fig. 7.1 is expanded to show its internal structure in Fig. 7.9. Each of the files in Fig. 7.1 is shown expanded into its constituent inode and data blocks. As an example of how these data structures work, consider how the system finds the file `/usr/bin/vi`. It must first search the root directory of the filesystem to find the directory `usr`. To do this, it must first find the inode that describes the root directory. By convention, inode 2 is always reserved for the root directory; therefore, the system finds and brings inode 2 into memory. This inode shows where the data blocks are for the root directory; these data blocks must also be brought into memory so that they can be searched for the entry `usr`. Having found the entry for `usr`, the system finds that the contents of `usr` are described by inode 4. Returning once again to the disk, the system fetches inode 4 to find where the data blocks for `usr` are located. Searching these blocks, it finds the entry for `bin`. The `bin` entry points to inode 7. Next the system brings in inode 7 and its associated data blocks from the disk, to begin searching for the entry for `vi`. Having found that `vi` is described by inode 9, the system can fetch this inode and discover the set of blocks that describe the `vi` binary.

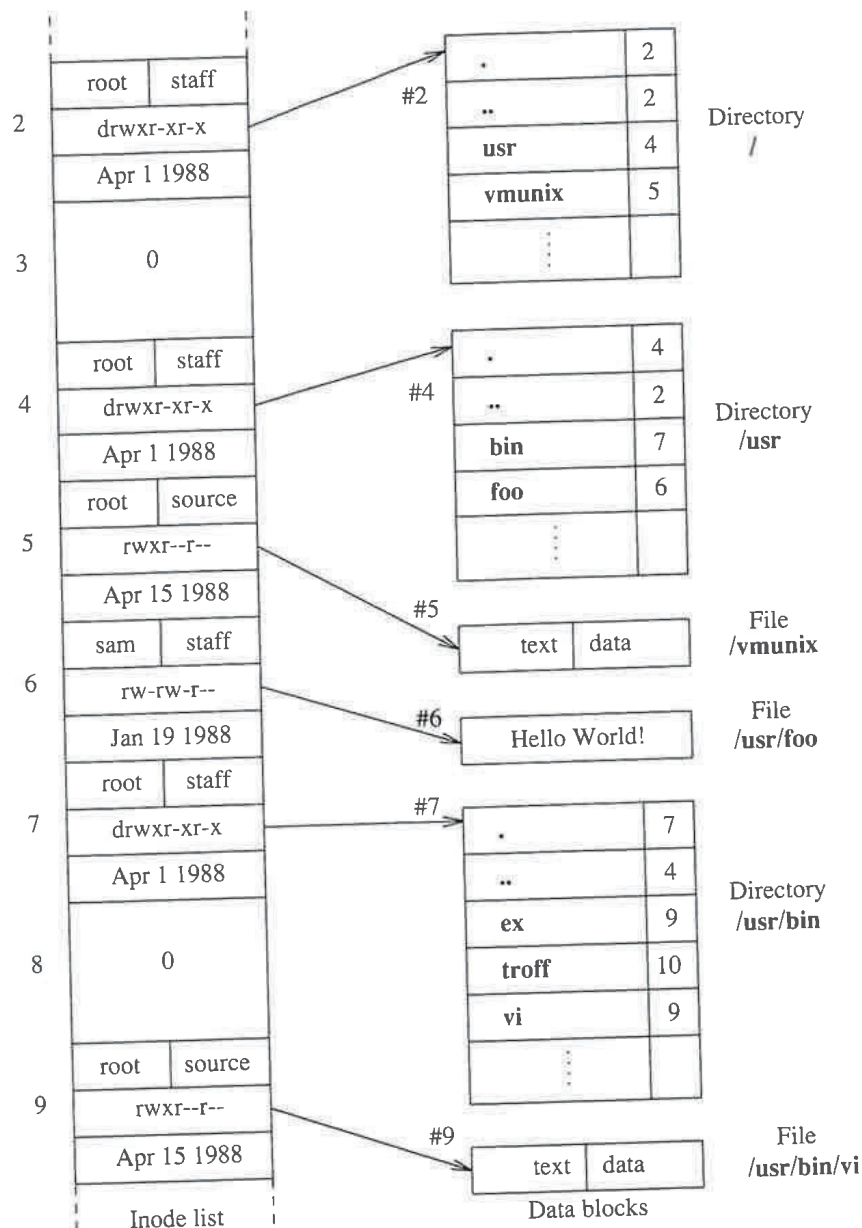


Figure 7.9 Internal structure of a small filesystem.

This whole process implies an enormous amount of disk I/O for even such trivial tasks as executing system utilities. To obtain reasonable performance, the system must cache frequently used information to reduce the number of disk requests. This section describes the inode table that holds recently used inodes, Section 7.5 discusses the system buffer pool that holds recently used disk blocks,

and Section 7.8 discusses the name-translation cache that holds recent name translations for the inodes held in the inode table. By using these caches, the system typically finds over 85 percent of the information that it needs without having to bring in the data from disk.

Inode Management

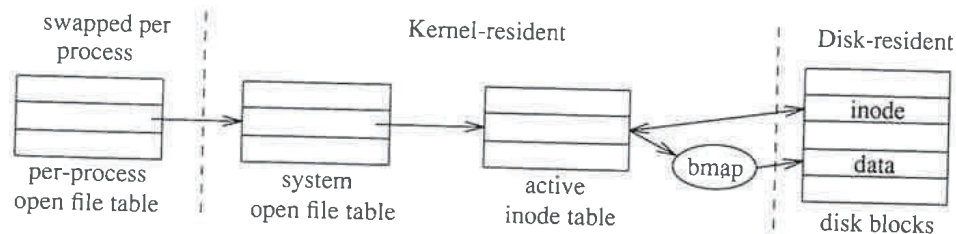
Most of the activity in the filesystem revolves around inodes. The kernel keeps a table of active and recently accessed inodes. Figure 7.10 shows the location of the inode table within the system.

As described in Section 6.2, each process has a *process open file table* that has slots for up to a reasonable number (such as 64) of file descriptors; this table is maintained as part of the process state. When a user process opens a file (or socket), an unused slot is located in its open file table; the small integer file descriptor that is returned on a successful *open* is the value that indexes into this table.

The per-process file-table entry points into the *system open file table*, each entry of which contains information about the underlying file or socket represented by the descriptor. For files, the *file table* points to the inode representing the open file. The active and recently used inodes are stored in another systemwide table called the *inode table*, shown in Fig. 7.11.

The first step in opening a file is to find the file's associated inode. First, the kernel searches the *inode table* to see whether the requested inode is already in memory. To avoid doing a linear scan of all the entries in the table, the system keeps a set of hash chains keyed on inode number and filesystem identifier. Even with an inode table containing hundreds of entries, the kernel needs to inspect only about five entries to determine whether a requested inode is already in the table. If the inode is not in the table, such as the first time a file is opened, the system must allocate a new inode-table slot and read in the inode from disk. The new inode-table slot is allocated from the least recently used inode in the inode table. To quickly find the oldest inode, the system keeps unused inodes linked together in an LRU chain. When the last reference to a file is closed, the system puts the inode onto the end of the LRU chain. Thus, the inode on the front of the LRU chain yields the least recently used inode. Frequently used inodes will be

Figure 7.10 Layout of kernel tables.



such
e, the
disk
inodes,
locks,

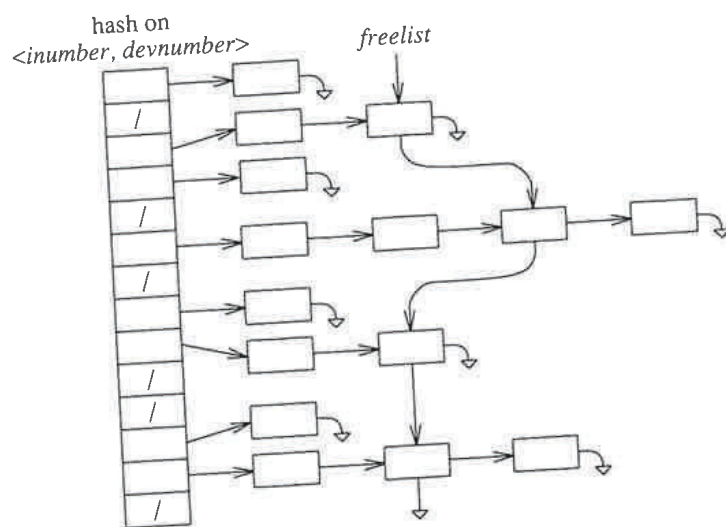


Figure 7.11 Structure of the inode table.

reclaimed from the middle of the LRU chain and will be replaced at the end after use. This algorithm reduces the probability of reuse of the inode slots.

The next step is to locate the disk block containing the inode and to read that block into a buffer in system memory. When the disk I/O completes, the inode is copied from the disk buffer into its slot in the inode table. In addition to the information contained in the disk portion of the inode, the inode table itself maintains supplemental information while the inode is in memory. This information includes the hash and use chains described previously, as well as flags showing the inode's status, reference counts on its use, and information to manage locks. The information also contains pointers to other kernel data structures of frequent interest, such as the superblock for the filesystem containing the inode.

Finding File Blocks

Once an inode has been brought into memory, the system can locate the data blocks associated with the file. It locates the physical data blocks associated with a file by interpreting the block pointers in the inode. The task of interpreting and allocating these pointers is managed by the *block-mapping* routine called *bmap()*. Figure 7.12 shows the procedural access path through the kernel. The pathname specified in an *open* call is passed to the name-to-inode translation routine *namei()*. This routine is responsible for doing the path traversal described previously, and ultimately for returning either a failure code or a pointer to an inode-table entry.

Having opened a file, a process can do reads or writes on it. These requests are channeled through the *rwip()* routine. *Rwip()* is responsible for converting the read or write into one or more reads or writes of logical file blocks. A logical

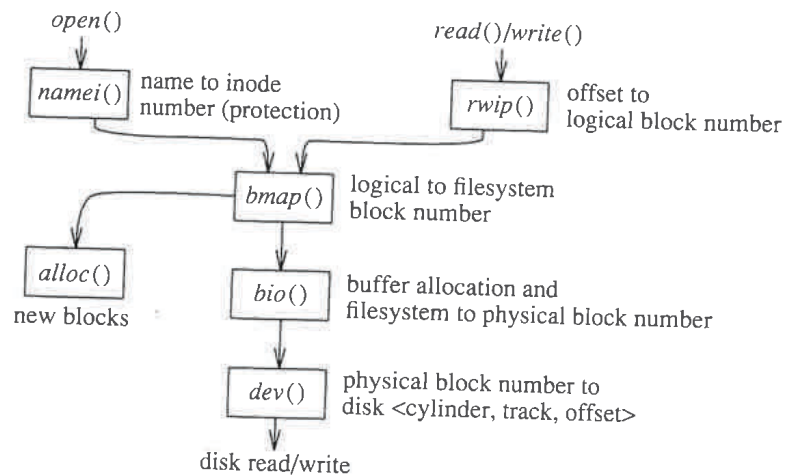


Figure 7.12 Procedural access through the filesystem.

block request is then handed off to *bmap()*. *Bmap()* is responsible for converting a logical block number to a physical block number by interpreting the direct and indirect block pointers in an inode. If a read is requested, *rwip()* requests the block I/O system to return a buffer filled with the contents of the disk block. If two or more logically sequential blocks are read from a file, the process is assumed to be reading the file sequentially. Here, *bmap()* returns two values, first the requested block, and second the number of the next block in the file. The system makes two disk requests, one for the requested block and a second for the read-ahead block in anticipation that the process will soon want that block.

If a write is requested, *rwip()* allocates a buffer to hold the contents of the block. The user's data are copied into the returned buffer, and the buffer is marked as dirty. If the buffer has been completely filled, it is queued to be written to the disk. If the buffer has not been completely filled, it is not immediately queued. The buffer is held in the expectation that the process will soon want to add more data to it. It is not released until it is needed for some other block; that is, until it has reached the head of the free list, or until a user process does a *sync* system call. There is normally a user process called *update* running that does a *sync* every 30 seconds.

File-Block Allocation

The task of managing block and fragment allocation is done by *bmap()*. If the file is being written and a block pointer is zero or points to a fragment that is too small to hold the additional data, *bmap()* calls the allocation routines to obtain a new block. If the file needs to be expanded, one of two conditions exists:

1. The file contains no fragmented blocks (and the last block in the file contains insufficient space to hold the new data). If space exists in a block already

allocated, the space is filled with new data. If the remainder of the new data consists of more than a full block, a full block is allocated and the first full block of new data is written there. This process is repeated until less than a full block of new data remains. If the remaining new data to be written will fit in less than a full block, a block with the necessary fragments is located; otherwise, a full block is located. The remaining new data are written into the located space. However, to avoid excessive copying for slowly growing files, the filesystem allows only direct blocks of files to refer to fragments.

2. The file contains one or more fragments (and the fragments contain insufficient space to hold the new data). If the size of the new data plus the size of the data already in the fragments exceeds the size of a full block, a new block is allocated. The contents of the fragments are copied to the beginning of the block and the remainder of the block is filled with new data. The process then continues as in step 1. Otherwise, a block big enough to hold the data is located; if enough of the rest of the current block is free, that block is used to avoid copying. The contents of the existing fragments, appended with the new data, are written into the allocated space.

Bmap() is also responsible for allocating blocks to hold indirect pointers. It must also deal with the special case in which a process seeks out past the end of a file and begins writing. Because of the constraint that only the last block of a file may be a fragment, *bmap()* must first ensure that any previous fragment has been upgraded to a full-sized block.

On completing a successful allocation (Section 7.7), the allocation routines return the block or fragment number to be used; *bmap()* then updates the appropriate block pointer in the inode. Having allocated a block, the system is ready to allocate a buffer to hold the block's contents so that the block can be written to disk.

7.5 Buffer Management

The semantics of the UNIX filesystem imply much disk I/O. If every implied disk transfer really had to be done, the CPU would spend most of its time waiting for I/O to complete. The task of the buffer cache is two-fold. One task is to manage the memory that buffers data being transferred to and from the disk. The second, and more important, task is to act as a cache of recently used disk blocks. On a typical UNIX system, over 85 percent of the implied disk transfers can be skipped because the requested block already resides in the buffer cache. Depending on available memory, a system may be configured with anything from a hundred to a thousand buffers. The larger the number of buffers, the longer a given disk block can be retained in memory, and the greater the chance that disk I/O can be avoided.

Figure 7.13 shows the format of a buffer. The buffer is composed of two parts. The first part is the buffer header, which contains information used to find

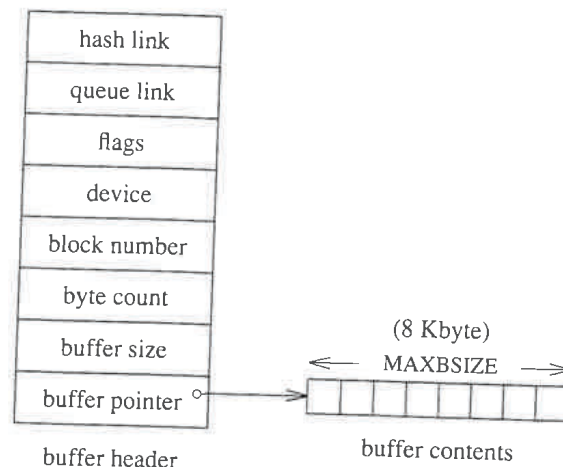


Figure 7.13 Format of a buffer.

the buffer and to describe its contents. The content information includes the device (i.e., disk and partition on that disk), the starting physical block number (counting from the beginning of the partition), and the number of bytes contained in the buffer. The flags entry tracks status information about the buffer, such as whether the buffer contains useful data, whether the buffer is in use, and whether the data must be written back to the disk before the buffer can be reused.

The second part is the actual buffer contents. Rather than the header being prepended to the data area of the buffer, as is done with mbufs (see Section 10.2), the data areas are maintained separately. Thus, there is a pointer to the data and a field that shows the size of the data buffer. The buffer size is always at least as big as the fragment size. Data are maintained separately from the header to allow easy manipulation of the buffer sizes with the page-mapping hardware. If the headers were prepended, either each header would have to be on a page by itself or the kernel would have to avoid remapping buffer pages that contained headers.

The size of buffer requests from the filesystem range from 512-byte fragments up to 8192-byte full-sized blocks. If many small files are being accessed, then many small buffers are needed. Alternatively, if several large files are being accessed, then fewer large buffers are needed. To allow the system to adapt efficiently to these changing needs, each buffer is allocated 8192 bytes of virtual memory, but the address space is not fully populated with physical memory. Initially, each buffer is assigned 2048 bytes of physical memory. As smaller buffers are allocated, they give up their excess physical memory to other buffers that need to hold more than 2048 bytes. The algorithms for managing the physical memory are described later.

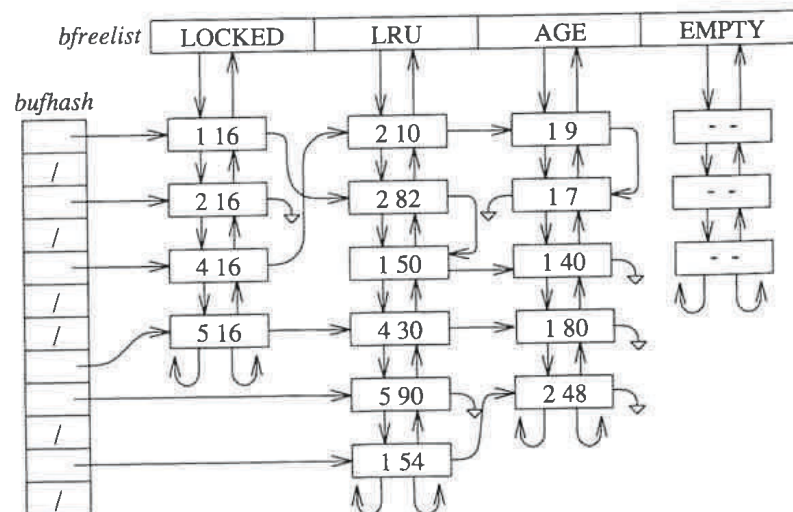
The internal kernel interface to the buffer pool is simple. The filesystem allocates and fills buffers by calling the *bread()* routine. *Bread()* takes a device, a block number, and a size, and returns a pointer to a locked buffer. Any other process that tries to access the buffer will be put to sleep until the buffer is released.

A buffer can be released in one of four ways. If the buffer has not been modified, it can simply be released through use of *brelse()*. If the buffer has been modified, it is called *dirty*. Dirty buffers must eventually be written back to the disk. Three routines are available based on the urgency with which the data must be written to disk. In the typical case, *bdwrite()* is used; it assumes that the buffer probably will be modified again soon, so should be marked as dirty, but should not be immediately written to disk. The heuristic is that, if the buffer will be modified again soon, the disk I/O would be wasted. Because the buffer is held for an average of 15 seconds before it is written, a process doing many small writes will not repeatedly access the disk. If the buffer has been completely filled, then it is unlikely to be written again soon, so it should be released with *bawrite()*. *Bawrite()* schedules a write to disk, but allows the caller to continue running while the output completes. The final case is *bwrite()*, which ensures that the disk write is complete before proceeding. Because this mechanism can introduce a long latency to the requester, it is used only when the process explicitly requests the behavior (such as the *fsync* system call), or when the operation is critical to ensure the consistency of the filesystem in case of a system crash.

Figure 7.14 shows a snapshot of the buffer pool. A buffer with valid contents is contained on exactly one *bufhash* hash chain. The hash chains are used to determine quickly whether a disk block is contained in the buffer pool, and, if the block is, to locate it. A buffer is removed only when it becomes invalid or is reused for a different block. Thus, even if the buffer is in use by one process, it can still be found by another process, although the *busy* flag will be set so that it will not be used until its contents are consistent.

In addition to the hash list, each buffer not currently in use appears on exactly one free list. The first free list is the locked list. Buffers on this list may not be

Figure 7.14 Snapshot of the buffer pool.



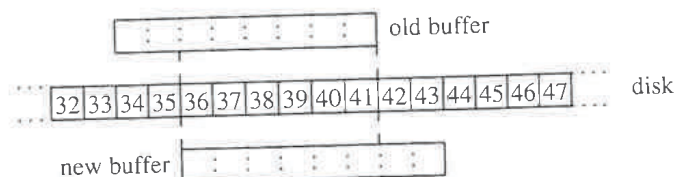
routines that implement the interface for getting buffers. The primary interface to getting a buffer is through *bread()*, which is called with a request for a data block of a specified size on a specified disk. *Bread()* first calls *getblk()* to find out whether the data block is available in a buffer that is already in memory. If the block is available in a buffer, *getblk()* calls *notavail()* to take the buffer off whichever free list it is on and to mark it busy; *bread()* can then return the buffer to the caller.

If the disk block is not already in memory, *getblk()* calls *newbuf()* to allocate a new buffer. The new buffer is then passed to *breadalloc()* to ensure that there will not be any overlap with other buffers. Next, the buffer is passed to *allocbuf()*, which ensures that the buffer has the right amount of physical memory. *Getblk()* then returns the buffer to *bread()* marked busy and unfilled. Noticing that the buffer is unfilled, *bread()* passes it to the disk driver to have the data read in. When the disk read completes, the buffer can be returned.

To maintain the consistency of the filesystem, *breadalloc()* ensures that a disk block is mapped into at most one buffer. If the same disk block were present in two buffers, and both buffers were marked dirty, the system would be unable to determine which buffer had valid information. Figure 7.16 shows a sample allocation. In the middle of the figure are the blocks on the disk. Above the disk is shown an old buffer containing a 4096-byte fragment for a file that presumably has been removed or shortened. The new buffer is going to be used to hold a 3072-byte fragment for a file that is presumably being created and that will reuse part of the space previously held by the old file. If the old buffer is marked dirty, it must first be written, then marked invalid, so that it cannot be found in the buffer pool again. After being marked invalid, the buffer is put at the front of the AGE queue so that it will be used before any buffers with potentially useful data. The system can then allocate the new buffer knowing that that buffer uniquely maps the corresponding disk blocks.

The final task in allocating a buffer is to ensure that the buffer has enough physical memory allocated to it; this task is handled by *allocbuf()*. Figure 7.17 shows the virtual memory for the data part of a buffer. The data area for each buffer is allocated 8192 bytes of virtual address space. The *bufsize* field in the buffer header shows how much of the virtual address space really is backed by physical memory. *Allocbuf()* compares the size of the intended data block with the amount of physical memory already allocated to the buffer. If there is excess

Figure 7.16 Potentially overlapping allocation of buffers.



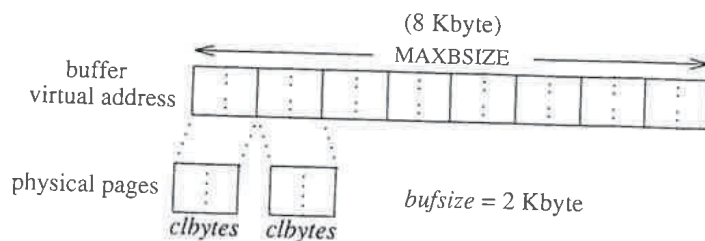


Figure 7.17 Allocation of buffer memory.

physical memory and there is a buffer available on the EMPTY queue, a buffer is taken off the EMPTY queue, the excess memory is put into the empty buffer, and it is then released onto the front of the AGE queue. If there are no buffers on the EMPTY queue, the excess physical memory is retained in the original buffer.

If the buffer has insufficient memory, *allocbuf()* takes memory from other buffers. *Allocbuf()* does the allocation by calling *newbuf()* to allocate another buffer and then transferring the physical memory in the new buffer to the buffer under construction. As before, if there is excess memory, it is released to the front of the AGE queue; otherwise, it is released to the EMPTY queue. If the new buffer still does not have enough physical memory, the process is repeated. *Allocbuf()* ensures that each physical-memory page is mapped into exactly one buffer at all times.

7.6 Quotas

The quota system allows administrators to control the number of inodes and data blocks that can be allocated by any user on each filesystem. Quotas for each user are maintained on a per-filesystem basis.

Quotas hook into the system primarily as an adjunct to the allocation routines. Figure 7.18 shows that, when the allocation routines are passed a request for a new block from *bmap()*, the request is first validated by *quotachk()*. *Quotachk()* consults the quota associated with the owner of the file. If the owner is below quota, its request is permitted and the request is added to their usage statistics. If the owner has reached or exceeded its limit, the request is denied and the allocator returns a failure to *bmap()* as though the filesystem were full.

Quotas are assigned to a filesystem after it has been mounted. A system call associates a file containing the quotas with the mounted filesystem. By convention, the file is named **quotas** and lives in the root of the mounted filesystem. The system opens the **quotas** file and holds a reference to it in the mount-table entry associated with the mounted filesystem; Fig. 7.19 shows the mount-table reference. As quotas for different users are needed, they can be fetched from the **quotas** file.

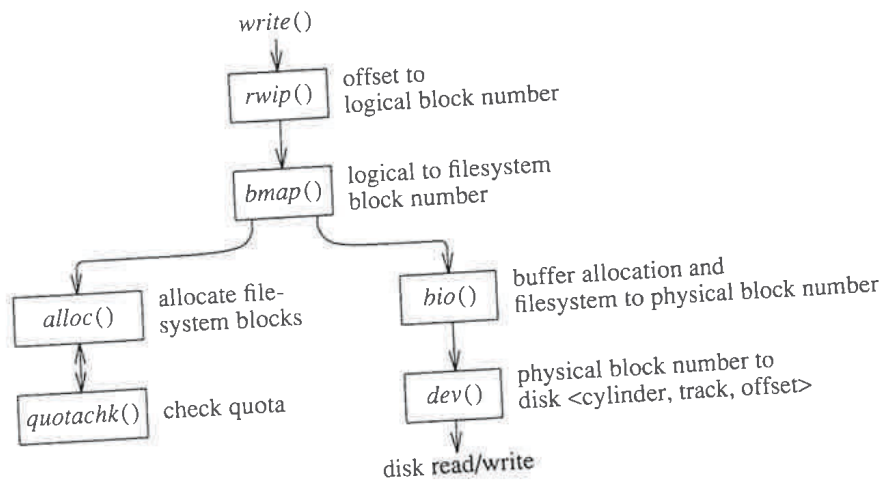
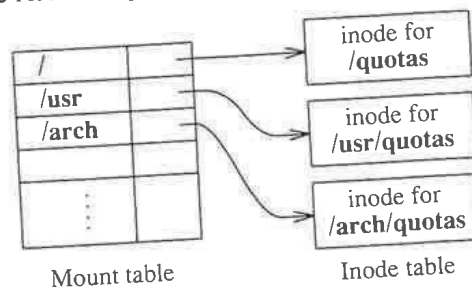


Figure 7.18 Procedural interface to quota control.

The **quotas** file is maintained as an array of quota records indexed by user identifiers; Fig. 7.20 shows a typical record. To find the quota for user identifier i , the system seeks to location $i \times \text{sizeof}(\text{quota structure})$ in the quota file and reads the quota structure at that location. Each **quota structure** contains the limits imposed on the user for the associated **filesystem**. These limits include the hard and soft limits on the number of blocks and inodes that the user may have, the number of blocks and inodes the user currently has allocated, and the number of warnings that the user has left before the soft limit is enforced as the hard limit.

Quotas are maintained in a system table called the *dquot table*; Fig. 7.21 shows two typical entries. As with the inode table, the dquot table maintains additional information about the quota information while that information is resident to allow faster access and identification. Since quotas may have to be updated on every write to a file, `quotachk()` must be able to find and manipulate them quickly. Thus, the task of finding the dquot structure associated with a file is done once

Figure 7.19 Access to quotas files.



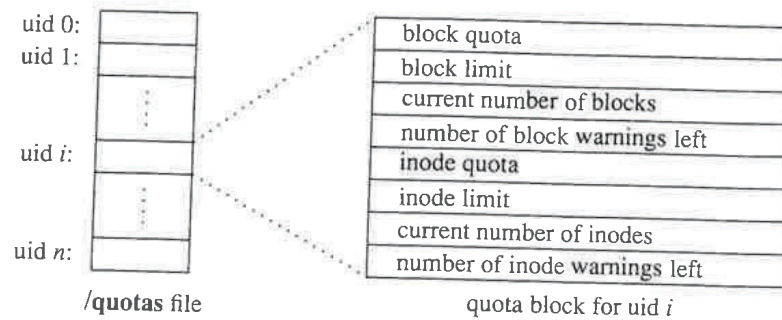
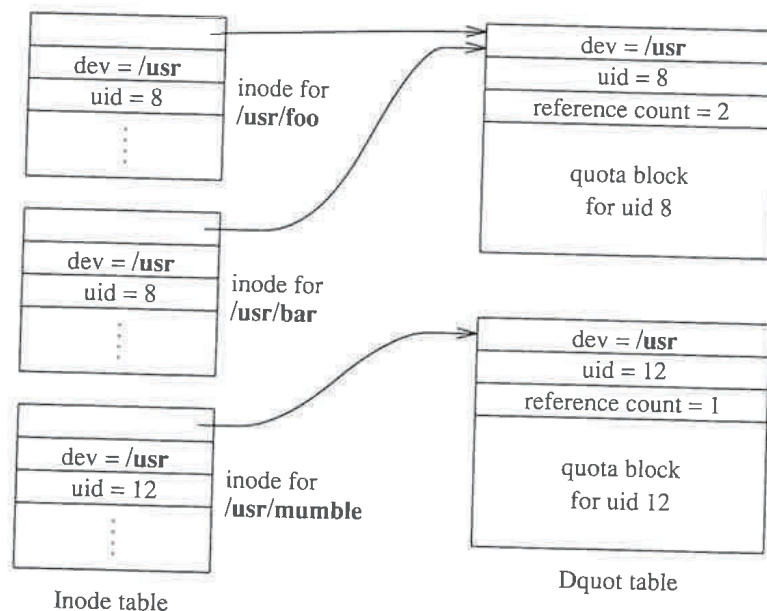


Figure 7.20 Contents of a quota record.

when the file is first opened. When the inode associated with a file is first entered into the inode table, the system checks to see whether there is a quota associated with it. If a quota exists, the inode is set up to hold a reference to the dquot structure as long as the inode is resident. The *quotchk()* routine can determine that a file has a quota by simply checking whether the dquot pointer is nonnull. If it is nonnull, all the necessary information can be accessed directly. If a user has multiple files open on the same filesystem, all inodes describing those files point to the same dquot entry. Thus, the number of blocks allocated to a particular user can always be easily and consistently known.

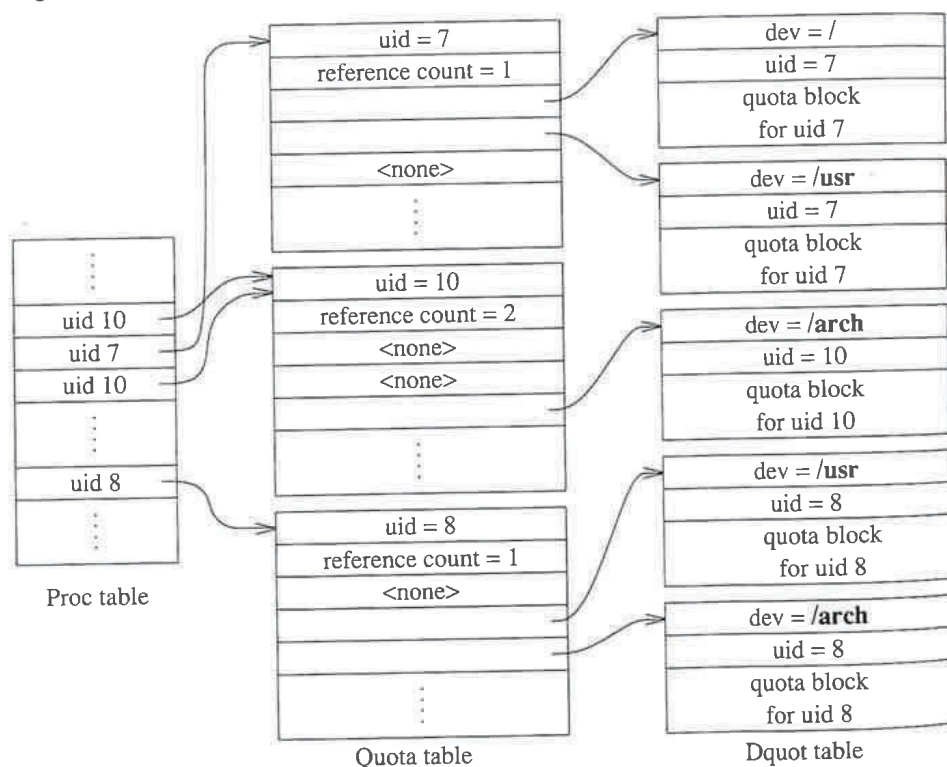
Figure 7.21 The dquot table.



The number of entries in the dquot table can grow large. To speed access, a hashed lookup scheme is used. In the dquot table, the hash is keyed on the filesystem and user identifier. The hashing structure and free list are set up in the same way as is the inode table shown in Fig. 7.11. When the reference count on a dquot structure drops to zero, the entry is linked onto the free list. The dquot structure is not removed from its hash chain, so if the structure is needed again soon it can still be located. Only when a dquot structure is recycled with a new quota record is it removed and relinked into the hash chain.

The hashing structure allows recently used dquot structures to be found quickly. However, it does not solve the problem of how to discover that a user has no quota on a particular filesystem. If a user has no quota, a lookup for the quota will fail. The cost of going to disk and reading the quota file to discover that the user has no quota imposed would be prohibitive. To avoid this work each time a new file is accessed, a *quota table* is maintained that has an entry per active user identifier; Fig. 7.22 shows a typical quota table. When an inode owned by a user that does not already have an entry in the quota table is first accessed, an entry for that user is made in the quota table. At that time, the system checks each filesystem that has quotas enabled to determine whether the user has quotas on

Figure 7.22 The quota table.



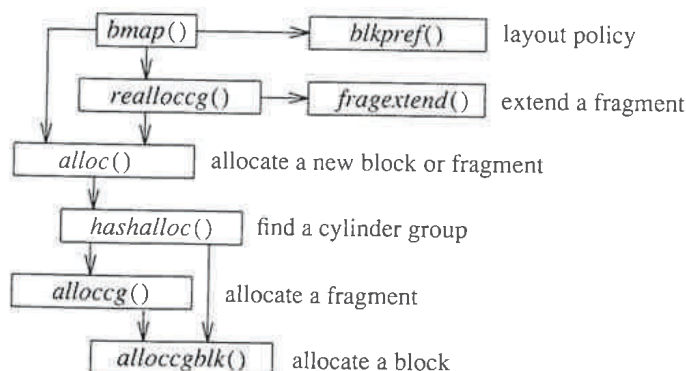
that filesystem. If quotas exist, a pointer to the dquot table is held for that filesystem. If quotas do not exist, the entry is marked to show that they do not. To further speed the usual case of users opening one of their own files, the system maintains a pointer from each proc-table entry to the quota-table entry for each user. When a process opens a file owned by the user running that process, the system follows the pointer to the structure for the user. If the user has a quota on the affected filesystem, the system can increment the reference count on the associated dquot structure and can store the reference in the inode. If the process opens a file of some other user, the system must hash into the quota table to find the entry for the other user; once that entry is found, the same procedure can be followed.

7.7 Allocation Mechanisms

We shall now turn our attention to the details of allocating a data block. The procedural description of the allocation process is shown in Fig. 7.23. *Bmap()* is the routine responsible for determining when a new block must be allocated. It first calls the layout-policy routine *blkpref()* to select the most desirable block based on the preference from the global-policy routines that were described in Section 7.3. If a fragment has already been allocated and needs to be extended, *bmap()* calls *realloccg()*. If nothing has been allocated yet, *bmap()* calls *alloc()*.

Realloccg() first tries to extend the current fragment in place. Consider the sample block of an allocation map with two fragments allocated from it, shown in Fig. 7.24. The first fragment can be extended from a size 2 fragment to a size 3 fragment, since an adjacent piece is unused. The second fragment cannot be extended, as it occupies a full block, and fragments are not allowed to span blocks. If *realloccg()* is able to expand the current fragment in place, the map is updated appropriately and it returns. If the fragment cannot be extended,

Figure 7.23 Procedural interface to block allocation.



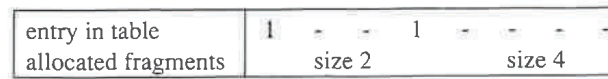


Figure 7.24 Sample block with two allocated fragments.

reallocg() calls the *alloc()* routine, described later, to get a new fragment. The old fragment is copied to the beginning of the new fragment, and the old fragment is freed.

The bookkeeping tasks of allocation are handled by *alloc()*. It first verifies that a block is available in the desired cylinder group by checking the filesystem summary information. If the summary information shows that the cylinder group is full, *alloc()* will quadratically rehash through the summary information looking for a cylinder group with free space. Having found a cylinder group with space, *alloc()* will call either the fragment-allocation routine or the block-allocation routine to acquire a fragment or block.

The block-allocation routine is given a preferred block. If that block is available, it is returned. If the block is unavailable, the allocation routine tries to find another block on the same cylinder that is rotationally closest to the requested block. To simplify the task of locating rotationally optimal blocks, the summary information for each cylinder group includes a count of the available blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a 3600 revolution-per-minute drive. The superblock contains a vector of lists called the *rotational-layout table*. The vector is indexed by rotational position. Each component of the vector lists the index into the block map for every data block contained in its rotational position. When looking for an allocatable block, the system first looks through the summary counts for a rotational position with a nonzero block count. It then uses the index of the rotational position to find the appropriate list of rotationally optimal blocks. This list enables the system to limit its scan of the free block map to only those parts that contain some free, rotationally well-placed blocks.

The fragment-allocation routine is given a preferred fragment. If that fragment is available, it is returned. If the requested fragment is not available, and the filesystem is configured to optimize for space utilization; it uses a best-fit strategy for fragment allocation. The fragment-allocation routine checks the cylinder-group summary information, starting with the entry for the desired size and scanning larger sizes until an available fragment is found. If there are no fragments of the appropriate size or larger, then a full-sized block is allocated and is broken up.

If an appropriate-sized fragment is listed in the fragment summary, then the allocation routine expects to find it in the allocation map. To speed up the process of scanning the potentially large allocation map, a table-driven algorithm is used. Each byte in the map is treated as an index into a *fragment-descriptor table*. Each entry in the fragment-descriptor table describes the fragments that are free for that corresponding map entry. Thus, by doing a logical AND with the bit

Bits in map	Decimal value
-111--11	115

Figure 7.25 Map entry for an 8192/1024 filesystem.

corresponding to the desired fragment size, the allocator can quickly determine whether the desired fragment is contained within a given allocation-map entry. As an example, consider the following entry from an allocation map for the 8192/1024 filesystem shown in Fig. 7.25. The map entry shown has already been fragmented, with a single fragment allocated at the beginning and a size 2 fragment allocated in the middle. Remaining unused is another size 2 fragment, and a size 3 fragment. Thus, if we look up entry 115 in the fragment table, we find the entry shown in Fig. 7.26. If we were looking for a size 3 fragment, we would inspect the third bit and find that we had been successful; if we were looking for a size 4 fragment, we would inspect the fourth bit and find that we needed to continue. The C code that implements this algorithm is as follows:

```
for (i = 0; i < MAPSIZE; i++)
    if (fragtbl[allocmap[i]] & (1 << (size - 1)))
        break;
```

Using a best-fit policy has the benefit of minimizing disk fragmentation; however, it has the undesirable property that it maximizes the number of fragment-to-fragment copies that must be made when a process writes a file in many small pieces. To avoid this behavior, the system can configure filesystems to optimize for time rather than for space. The first time a process does a small write on a filesystem configured for time optimization, it is allocated a best-fit fragment. On the second small write, however, a full-sized block is allocated, with the unused portion being freed. Later small writes are able to extend the fragment in place, rather than requiring additional copy operations. Under certain circumstances, this policy can cause the disk to become heavily fragmented. The system tracks this condition, and automatically reverts to optimizing for space if the fragmentation gets within 2 percent of the minimum free-space limit.

7.8 Translation of Filesystem Names

Profiling studies show that nearly one-quarter of the time in the kernel is spent in the pathname-translation routine, *namei()*, translating pathnames to inodes. *Namei()* is responsible for doing access checks on each component of the name to verify that the current process has the necessary permissions to search the intervening directories, and to access the target file in the way that the process has requested. *Namei()* consists of two nested loops. The outer loop is traversed once per pathname component. The inner loop performs a linear search through a directory looking for a particular pathname component.

entry in table	0	0	0	0	0	1	1	0
available fragment size	8	7	6	5	4	3	2	1

Figure 7.26 Fragment-table entry for entry 115.

Two caching techniques are used to reduce the cost of name translation. The first technique is to reduce the number of iterations around the inner loop of *namei()* by observing that many programs sequentially scan a directory. To improve performance for processes exhibiting this behavior, the directory offset of the last component of the most recently translated pathname is maintained for each process. If the next name that the process requests to be translated is in the same directory, the search is started from the offset at which the previous name was found (instead of from the beginning of the directory). Changing directories invalidates the cache. For programs that step sequentially through a directory with n files, search time decreases from $Order(n^2)$ to $Order(n)$.

A quick benchmark to show the maximum effectiveness of the cache is *ls -l* on a directory containing 600 files. Adding the per-process cache reduces the amount of system time for this test by 85 percent. Unfortunately, the maximum effectiveness is much greater than the average effectiveness. Although the cache is 90 percent effective when hit, it is usable on only about 25 percent of the names being translated. An additional reason for the improvement being small is that, although the amount of time spent in *namei()* itself decreases substantially, more time is spent in the routines that it called, because cache misses cause a directory to be accessed twice—once to search from the middle to the end, and once to search from the beginning to the middle.

Frequent requests for a small set of names are best handled with a cache of recent name translations. 4.3BSD uses a cache keyed on a name and the inode and device number of the directory that contains the name. Associated with each entry is a pointer to the corresponding entry in the inode table. This cache has the effect of eliminating the inner loop of *namei()*. For each pathname component, *namei()* first looks in its cache of recent translations for the needed name. If the name exists, the directory search can be eliminated.

An issue to be considered is how the name cache should hold references to the inode table. Normally, processes hold *hard references* by incrementing the reference count in the inode they reference. Since the system reuses only inodes with zero reference counts, a hard reference ensures that the inode pointer will remain valid. However, if the name cache holds hard references, its size is limited to some fraction of the size of the inode table, since some inodes must be left free for new files. Holding of hard references also makes it impossible for other parts of the kernel to verify sole use of a device or file. These are reasons why it is impractical to use hard references without affecting the behavior of the inode-caching scheme. Thus, *soft references* are kept, protected by a *capability*—a 32-bit number guaranteed to be unique. When all the numbers have been exhausted, all outstanding capabilities are purged, and numbering restarts from scratch.

Purging is possible, as all capabilities are easily found in kernel memory. When an entry is made in the name cache, the capability of its inode is copied to the name-cache entry. When an inode is reused, it is issued a new capability. When a name-cache hit occurs, the capability of the name-cache entry is compared with the capability of the inode that the entry references. If the capabilities do not match, the name-cache entry is invalid. Since the name cache holds only soft references, it may be sized independent of the size of the inode table. A final benefit of using capabilities is that all cached names for an inode can be invalidated without a search through the entire cache; instead, it is necessary only to assign a new capability to the inode.

In 1984, on a general time-sharing system at Berkeley during the 12-hour period from 8:00 A.M. to 8:00 P.M., the system did 500,000 to 1,000,000 name translations. Statistics on the performance of both caches showed that the large performance improvement was caused by the high hit ratio. The name cache had a hit rate of 70 to 80 percent; the directory offset cache hit rate was 5 to 15 percent. The combined hit rate of the two caches almost always was 85 percent. With the addition of the two caches, the percentage of system time devoted to name translation dropped from 25 percent to less than 10 percent. Although the systemwide cache reduced both the amount of time in the routines that *namei()* calls and the amount in *namei()* itself (since fewer directories needed to be accessed or searched), it is interesting to note that the actual percentage of system time spent in *namei()* itself increased even though the actual time per call decreased. This increase occurred because less total time was being spent in the kernel; hence, a smaller absolute time became a larger total percentage.

Exercises

- 7.1 Why are directory entries not allowed to span chunks?
- 7.2 Why are hard links not permitted to span filesystems?
- 7.3 Describe the difference in interpretation of a symbolic link containing an absolute pathname from that of a symbolic link containing a relative pathname.
- 7.4 Explain why unprivileged users are not permitted to make hard links to directories, but are permitted to make symbolic links to directories.
- 7.5 How can hard links be used to gain access to files that could not be accessed if a symbolic link were used instead?
- 7.6 How does the system recognize loops caused by symbolic links? Suggest an alternative scheme for doing loop detection.
- 7.7 How many blocks and fragments are allocated to a 31,200-byte file on a filesystem with 4096-byte blocks and 1024-byte fragments? How many blocks and fragments would be allocated to this file on a filesystem with

- 4096-byte blocks and 512-byte fragments? Answer these two questions assuming that an inode had only six direct block pointers, instead of 12.
- 7.8 What is the difference between a logical block and a physical block? Why is this distinction important?
- 7.9 How do quotas differ from the file-size resource limits described in Section 3.8?
- 7.10 Why is the **per-cylinder** group information placed at varying offsets from the beginning of the cylinder group?
- 7.11 Why are triple indirect blocks not **commonly useful** on machines with a 32-bit word size and a 512-byte minimum **fragment size**?
- 7.12 What is a quadratic hash? Describe what it is used for in the filesystem, and why it is used for that purpose.
- 7.13 Explain why the 4.3BSD filesystem maintains a 5 to 10 percent reserve of free space. What problems would arise if the free-space reserve were set to zero?
- 7.14 Why are the allocation policies for inodes different from those for data blocks?
- 7.15 Why are the buffer headers allocated separately from the memory that holds the contents of the buffer?
- 7.16 How does the maximum filesystem block size affect the buffer cache?
- 7.17 Give two reasons why increasing the basic block size in the old filesystem from 512 bytes to 1024 bytes more than doubled the system's throughput.
- 7.18 Describe the two methods used to improve the performance of the traditional UNIX filesystem in 4.2BSD.
- *7.19 Why is there an AGE list and an LRU list, instead of all buffers being managed on the LRU list?
- *7.20 **Filenames** can be up to 255 characters long. How can the systemwide **name cache** be implemented to avoid allocating 255 bytes for each entry?
- *7.21 If a process reads a **large file**, the blocks of the file will fill the buffer cache completely, **flushing out all other contents**. All other processes in the system then will have to go to disk for all their filesystem accesses. Describe an algorithm to control the purging of the buffer cache.
- *7.22 Discuss the tradeoff between dedicating memory to the buffer cache and making the memory available to the virtual-memory system for use in fulfilling paging requests. Give a policy for moving memory between the buffer pool and the virtual-memory system.

- *7.23 Which filesystem operations must be done synchronously to ensure that the filesystem can always be recovered deterministically after a crash (barring unrecoverable hardware errors)?
- *7.24 What problems would arise if files had to be allocated in a single contiguous piece of the disk? Consider the problems created by multiple processes, random access, and files with holes.
- **7.25 Inodes could be allocated dynamically as part of a directory entry. Instead, inodes are statically allocated when the filesystem is created. Why is the latter approach used?

References

- Elz, 1984.
K. R. Elz, "Resource Controls, Privileges, and Other MUSH," *USENIX Association Conference Proceedings*, pp. 183-191 (June 1984).
- Knuth, 1975.
D. Knuth, *The Art of Computer Programming, Volume 3—Sorting and Searching*, Addison-Wesley, Reading, MA (1975).
- McKusick & Kowalski, 1986.
M. K. McKusick & T. J. Kowalski, "Fck—The UNIX File System Check Program," pp. 5:1-22 in *UNIX System Manager's Manual, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version*, USENIX Association, Berkeley, CA (1986).
- Nevalainen & Vesterinen, 1977.
O. Nevalainen & M. Vesterinen, "Determining Blocking Factors for Sequential Files by Heuristic Methods," *The Computer Journal* **20**(3), pp. 245-247 (August 1977).
- Trivedi, 1980.
K. Trivedi, "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments," *Journal of the ACM* **27**(3), pp. 457-473 (July 1980).

Device Drivers

A *hardware device* is a peripheral such as a disk or tape drive, terminal multiplexer, or network controller. For each hardware device supported by UNIX, a software module termed a *device driver* is required. Device drivers provide a consistent interface to varying hardware devices by hiding device-specific details from the rest of the UNIX kernel and user applications.

UNIX provides three internal interfaces that a device driver may support. These interfaces permit a device to be used in three different ways: as a block-oriented device suitable for holding one or more filesystems, as a message-oriented channel through which network communication may take place, or as an unstructured device that is potentially suitable for use by the terminal I/O system. A device driver may provide any or all of these interfaces to the system. Most device drivers provide one interface, although some device drivers provide more. This chapter examines how UNIX supports hardware devices and how device drivers operate within the framework of the I/O system. The network-related aspects of handling hardware devices are discussed in Chapter 11, together with the networking facilities.

8.1 Overview

Computers store and retrieve data through supporting peripheral I/O devices. These devices typically include mass-storage devices, such as moving-head disk drives, magnetic-tape drives, and network interfaces. Storage devices such as disks and tapes are accessed through I/O controllers that manage the operation of their *slave* devices according to I/O requests from the CPU.

On the VAX machines supported by 4.3BSD, most peripheral devices are connected to the *UNIBUS* or to the *MASSBUS*. These buses are, in turn, connected to a VAX's main system bus through *bus adapters*. The UNIBUS and MASSBUS are

relatively old buses that were originally designed for PDP-11 computers. DEC used bus adapters to these buses in the original VAX machines to provide backward hardware compatibility and to allow customers to upgrade their hardware systems while still using existing peripherals.

I/O devices are accessed through memory locations in the physical address space of the CPU. That is, the VAX uses a *memory-mapped I/O architecture*. Since the UNIX kernel runs with the memory management enabled, the physical addresses associated with hardware devices must be *mapped* into the virtual address space of the kernel.

In addition to mapping the memory locations at which I/O devices reside, the system must also be prepared to field *interrupts* from devices and to invoke the appropriate device-driver software to *service* the interrupt. All UNIBUS and MASSBUS I/O devices utilize *vectored interrupts*. That is, when a device needs to interrupt the CPU, it transmits an *interrupt vector* along with its interrupt request. When the interrupt request is honored, the interrupt vector is used to index into an *interrupt-vector table* to extract the address of the interrupt service routine that should process the interrupt. For each I/O device present on a machine, UNIX must initialize the appropriate entries in the interrupt-vector tables. As we shall see later, the interrupt service routines for a device driver are not invoked directly from the interrupt-vector tables. Instead, UNIX interposes a small layer of code that saves machine state and establishes *context* for the device driver. This context includes the identity of the device that is interrupting.

Under UNIX, applications cannot directly access most I/O devices. Instead, the system interposes one or more layers of software between the device and the application to promote a uniform interface across all devices of the same type. For example, disk drives are typically accessed only through the filesystem, whereas network interface controllers are utilized only through the interprocess-communication facilities described in Chapter 10. In some cases, however, user applications need direct, or almost direct, access to I/O devices. In these instances, UNIX supports two standard interfaces to hardware devices through block and character special device files. Devices that are accessed through a block (character) device special file are referred to as *block devices* (*character devices*) or as having a *block-device interface* (*character-device interface*).

A block-device interface, as the name indicates, supports only block-oriented I/O operations. The block-device interface uses the buffer cache to minimize the number of I/O requests that actually require an I/O operation, and to synchronize with filesystem operations on the same device. All I/O is done to or from I/O buffers that reside in the kernel's address space. This approach requires at least one memory-to-memory copy operation to satisfy a user request, but also allows UNIX to support I/O requests of nearly arbitrary size and alignment.

A character-device interface comes in two styles that depend on the characteristics of the underlying hardware device. For some character-oriented hardware devices, such as terminal multiplexers, the interface is truly character-oriented, although higher-level software such as the terminal driver may provide a line-oriented interface to applications. However, for block-oriented devices such

as disks and tapes, a character device interface is actually a very *unstructured* or *raw* interface. For this interface, I/O operations do not go through the buffer cache; instead, they are made directly between the device and buffers in the application's virtual address space. Consequently, the size of the operations must be a multiple of the underlying *block size* required by the device and, on some machines, the application's I/O buffer must be aligned on a suitable boundary.

Internal to the system, I/O devices are accessed through a fixed set of entry points provided by each device's *device driver*. The set of entry points varies according to whether the I/O device supports a block- or character-device interface. For a block-device interface, a device driver is described by a *bdevsw* structure, whereas for character-device interface, it accesses a *cdevsw* structure. All the *bdevsw* structures are collected in the *block-device table*, whereas *cdevsw* structures are similarly organized in a *character-device table*.

Devices are identified by a *device number* that is constructed from a *major* and a *minor* device number. The *major device number* uniquely identifies the type of device (really the device driver) and is the index of the device's entry in the *block- or character-device table*. Devices that support both block- and character-device interfaces have two major device numbers, one for each table. The *minor device* number is interpreted solely by the device driver and is used by the driver to identify to which, of potentially many, hardware devices an I/O request refers. For magnetic tapes, for example, minor device numbers are used to identify a specific controller and tape transport. The minor device number may also specify a section of a device; for example, a channel of a multiplexed device, or optional handling parameters.

The remainder of this chapter describes the structure and operation of device drivers. The next section examines the general structure of a device driver and discusses the data structures shared between the I/O system and device drivers. Following this examination, the interaction between device drivers and the autoconfiguration facilities is considered. Finally, the operation of device drivers for hardware devices that reside on a UNIBUS and a MASSBUS are discussed.

8.2 Device Drivers

A device driver is divided into three main sections:

1. Autoconfiguration and initialization routines
2. Routines for servicing I/O requests (the top half)
3. Interrupt service routines (the bottom half)

The autoconfiguration portion of a driver is responsible for *probing* for a hardware device to see whether the latter is present and to initialize the device and any associated software state that is required by the device driver. This portion of the driver is typically called only once, when the system is initialized.

The section of a driver that services I/O requests by the system is invoked as a result of system calls or on behalf of the virtual-memory system. This portion of the device driver executes synchronously in the top half of the kernel and is permitted to block by calling the *sleep()* routine. We commonly refer to this body of code as the *top half* of a device driver.

Interrupt service routines are invoked when the system fields an interrupt from a device. Consequently, these routines cannot depend on any per-process state and cannot block. On the VAX, interrupt routines execute on the interrupt stack. We commonly refer to a device driver's interrupt services routines as the *bottom half* of a device driver.

In addition to these three sections of a device driver, an optional *crash-dump* routine may be provided. This routine, if present, is invoked when the system recognizes an unrecoverable error and wishes to record the contents of physical memory for use in postmortem analysis. Most device drivers for disk controllers, and some for tape controllers, provide a crash-dump routine. The use of the crash-dump routine is described in Chapter 13.

I/O Queueing

Device drivers typically manage one or more queues of I/O requests in their normal operation. As an input or output request is received by the top half of the driver, it is recorded in a data structure that is placed on a per-device queue for processing. When an input or output operation completes, the device driver receives an interrupt from the controller. The interrupt service routine removes the appropriate request from the device's queue, notifies the requester that the command has completed, and then initiates another request from the queue. The I/O queues are the primary means of communication between the *top* and *bottom* halves of a device driver. Block-oriented devices construct their I/O queues from *buf* structures, whereas character-oriented devices use *C-list* structures.

As I/O queues are shared among asynchronous routines, access to the queues must be synchronized. Routines that make up the top half of a device driver must raise the processor priority level (using *splbio()*, *splty()*, etc.) to prevent the bottom half from being entered as a result of an interrupt while a top-half routine is manipulating an I/O queue. Synchronization among multiple processes initiating I/O requests must also be done. This synchronization is done using the mechanisms described at the end of Chapter 4.

In addition to per-device I/O queueing, on the VAX, device drivers must also interact with the support routines for the bus adapter on which a device resides. For example, a device driver for a UNIBUS-based disk controller must schedule access to the UNIBUS through the UNIBUS adapter-support routines. This scheduling is required because limited resources are available in each adapter for doing I/O operations. In addition, certain I/O devices require exclusive use of the bus for reliable operation. To handle these problems, device drivers package each I/O request, then pass the request to a bus-adapter routine that schedules the I/O operation. When resources are available for the operation to be done, the device driver is *called back* and the I/O operation is started. This scheduling may involve additional queueing beyond that provided by the device driver.

Interrupt Handling

Interrupts are generated by devices to signal that an operation has completed or that a change in status has occurred. On receiving a device interrupt, the system invokes the appropriate device-driver interrupt service routine with one or more parameters that uniquely identify the device that requires service. These parameters are needed because device drivers typically support multiple devices of the same type. If the interrupting device's identity were not supplied with each interrupt, the driver would be forced to poll all the potential devices to identify the device that interrupted.

Interrupt routines for UNIBUS devices are passed a *physical unit number* that is defined at the time the system is configured. Interrupt routines for MASSBUS devices are actually invoked indirectly from the MASSBUS adapter interrupt service routines. The MASSBUS adapter interrupt service routines are passed a *MASSBUS unit number* that is defined at the time the system is configured. The system arranges, in each case, for the appropriate parameter to be passed to the interrupt service routine for each device by installing the address of an auxiliary glue routine in the interrupt vector table. This glue routine, rather than the actual interrupt service routine, is invoked to service the interrupt; it takes the following actions:

1. Save all volatile registers
2. Update statistics on device interrupts
3. Call the interrupt service routine with the appropriate unit number parameter
4. Restore the volatile registers saved in step 1
5. Return from the interrupt

Because a glue routine is interposed between the interrupt-vector table and the interrupt service routine, device drivers do not need to be concerned with saving and restoring machine state. In addition, special-purpose instructions that cannot be generated from C, but that are needed by the hardware to support interrupts, can be kept out of the device driver; this permits device drivers to be written without assembly language.

8.3 Block Devices

Device drivers for block devices are described by an entry in the *bdevsw* table. Block devices are most commonly associated with disk drives and tape transports. Each *bdevsw* structure contains the following entry points:

open Open the device in preparation for I/O operations. A device's open entry point will be called for each *open* system call on a block special device file, or, internally, when a device is prepared for mounting a filesystem with the *mount* system call. The *open()* routine will commonly verify the integrity of the associated medium. For example, it will verify that the device was identified during the

autoconfiguration phase and, for tape and disk drives, that a medium is present and on-line.

- strategy* Initiate a read or write operation and return **immediately**. I/O requests to or from filesystems located on a device are **translated** by the system into calls to the block I/O routines *bread()* and *bwrite()*. These block I/O routines in turn call the device's strategy routine to read or write data not in the cache. Each call to the strategy routine specifies a pointer to a *buf* structure containing the parameters for an I/O request. If the request is **synchronous**, the caller must sleep (on the address of the *buf* structure) until I/O completes.
- close* Close a device. The *close()* routine is called after the last client interested in using the device terminates. These semantics are defined by the **higher-level** I/O facilities. Disk devices have nothing to do when a **device** is closed, and thus use a null *close()* routine. Devices that support access to only a single client must mark the device as available once again. Closing a tape drive that was open for writing typically causes end-of-file marks to be written on the tape and the tape to be rewound.
- dump* Write all of physical memory to the device. The dump entry point is used to save the contents of memory on secondary storage. The system automatically takes a dump when it detects an unrecoverable error and is about to *crash*. The dump is used in a postmortem analysis of the problem that caused the system to crash. The dump routine is invoked with the processor priority at its highest level; thus, the device driver must poll for device status, rather than wait for interrupts. All disk devices are expected to support this entry point; some tape devices do as well.
- psize* Return the size of a disk-drive partition. The driver is supplied a logical unit and is expected to return the size of that unit, typically a disk-drive partition, in *DEV_BSIZE* blocks. This entry point is used during the bootstrap procedure to calculate the location at which a crash dump should be placed and to determine the sizes of the swap devices.

8.4 Character Devices

Character devices are described by entries in the *cdevsw* table. The entry points in this table (see Table 8.1) are used to support **raw** access to block-oriented devices as well as normal access to character-oriented devices through the terminal driver. Because of the diverse requirements of these two types of devices, the set of entry points is really the union of two disjoint sets. Raw devices support a subset of the entry points that correspond to those entry points found in a block device driver, whereas character devices support the full set of entry points. Each is described in the following sections.

Table 8.1 Entry points for character and raw device drivers.

Entry point	Function
<i>open()</i>	open the device
<i>close()</i>	close the device
<i>ioctl()</i>	perform an I/O control operation
<i>mmap()</i>	map device contents into memory
<i>read()</i>	perform an input operation
<i>reset()</i>	reinitialize device after a bus reset
<i>select()</i>	poll device for I/O readiness
<i>stop()</i>	stop output on the device
<i>write()</i>	perform an output operation

Raw Devices and Physical I/O

Most raw devices differ from block devices only in the way they do I/O. Whereas block devices read and write data to and from the system buffer cache, raw devices transfer data to and from user data buffers. Bypassing the buffer cache eliminates the memory-to-memory copy that must be done by block devices, but also denies applications the benefits of data caching. In addition, for devices that support both raw- and block-device access, applications must take care to preserve consistency between data in the buffer cache and data written directly to the device; the raw device should be used only when the block device is idle. Raw-device access is used by many filesystem utilities such as the filesystem check program, **fsck**, and by programs that read and write magnetic tapes—for example, **tar**, **dump**, and **restore**.

Because raw devices bypass the buffer cache, they are responsible for managing their own buffer structures. Each raw-device driver maintains a static set of buffer structures, one for each unit on which raw I/O is supported. The read and write routines use the *physio()* routine to initiate a raw I/O operation (see Fig. 8.1). The *strategy* parameter identifies a block-device strategy routine that is used to initiate I/O operations on the device. The buffer indicated by *bp* is used by *physio()* in constructing the request(s) made to the strategy routine. The device, read-write flag, and *uio* parameters completely specify the I/O operation that should be done. The *minphys()* routine is called by *physio()* to adjust the size of each I/O transfer before the latter is passed to the strategy routine; this allows the transfer to be done in sections, according to the maximum transfer size supported by the device.

The operation of *physio()* includes synchronizing access to the buffer structure used in doing the I/O operation. This synchronization is required because each device maintains only a single buffer for doing raw I/O operations. There are checks to ensure that only one process at a time will execute the section of code using the raw buffer. There is no other particular reason to restrict raw I/O

```

physio(strategy, bp, dev, flags, minphys, uio)
{
    int strategy();
    buffer *bp;
    device dev;
    int flags;
    int minphys();
    struct uio *uio;

    check user read/write access to the data buffer;
    raise the processor priority to splbio;
    while (the buffer is marked busy) {
        mark the buffer wanted;
        wait until the buffer is available;
    }
    lower the priority level;
    set up the fixed part of the buffer for a transfer;
    while (there are data to transfer and no I/O error) {
        mark the buffer busy for physical I/O;
        set up the buffer for a maximum sized transfer;
        call minphys to bound the transfer size;
        lock the part of the user address space
            involved in the transfer;
        call strategy to start the transfer;
        raise the priority level to splbio;
        wait for the transfer to complete;
        unlock the part of the address space previously
            locked;
        lower the priority level;
        deduct the transfer size from the total number
            of data to transfer;
    }
    clean up the state of the buffer;
    if (another process is waiting for the raw I/O buffer)
        wake up processes waiting to do physical I/O;
}

```

Figure 8.1 Algorithm for physical I/O.

operations for each device to be single threaded since, as we shall see, the device drivers queue I/O operations to a controller as needed. The 4.3BSD Tahoe release avoids this single threading of raw I/O requests by borrowing swap buffer structures instead of using per-device buffers.

Raw-device I/O operations request the hardware device to transfer data directly to or from the data buffer in the user program's address space described by the *uio* parameter. Thus, unlike I/O operations that do direct memory access

(DMA) from buffers in the kernel address space, raw I/O operations must check that the user's buffer is accessible by the device, and must lock it into memory for the duration of the transfer.

Character-Oriented Devices

Character-oriented I/O devices are typified by terminal multiplexers, although they also include printers and other character- or line-oriented devices. These devices are usually accessed through the terminal driver, described in Chapter 9. The close tie to the terminal driver has heavily influenced the structure of character device drivers. For example, several entry points in the *cdevsw* structure exist for communication between the generic terminal handler and the terminal multiplexer hardware drivers. This interface is described fully in Chapter 9.

Entry Points for Character Device Drivers

A device driver for a character device is defined by an entry in the *cdevsw* table. This structure contains many of the same entry points found in an entry in the *bdevsw* table.

- open* The *open()* and *close()* entry points provide similar functions to those of a block device driver. In fact, for character devices that simply provide raw access to a block device, these entry points are usually the same. But some block devices do not have these entry points, whereas most character devices do have them.
- close*
- read* Read data from a device. For raw devices, this entry point normally just calls the *physio()* routine with device-specific parameters. For terminal-oriented devices, a read request is passed immediately to the terminal driver. For other devices, a read request requires that the specified data be copied into the kernel's address space, typically with the *uiomove()* routine, and then be passed to the device.
- write* Write data to a device. This entry point is a direct parallel of the read entry point: Raw devices utilize *physio()*, terminal-oriented devices call the terminal driver to do this operation, and other devices handle the request internally.
- ioctl* Perform an operation other than a read or write. This entry originally provided a mechanism to get and set device parameters for terminal devices, and its use has expanded to other types of devices as well. Historically, *ioctl()* operations have varied widely from device to device. 4.3BSD, however, defines a set of operations that are supported by all tape devices. These operations position tapes, return unit status, write end-of-file marks, and place a tape drive off-line.
- select* Check the device to see whether data are available for reading and/or space is available for writing data. The select entry point is used by the *select* system call in checking file descriptors associated with device

special files. For raw devices, a select operation is meaningless, since data are not buffered. In this case, the entry point is set to *seltrue()*, a routine that returns true for any select request. For devices used with the terminal driver, this entry point is set to *ttselect()*, a routine described in Chapter 9.

- stop* Stop output on a device. The stop routine is meaningful only for devices used with the terminal driver. For these devices, the stop routine is used to halt transmission on a line when the terminal driver receives a *stop character*—for example, “^S”—or when it prepares to flush its output queues.
- mmap* Map a device’s contents into memory. This entry point was added in preparation for the *mmap* system call; its semantics are currently undefined.
- reset* Reset device state after a bus reset. The reset routine is called from the bus-adapter support routines after a bus reset is made. The device driver is expected to reinitialize the hardware, and to reset any driver state related to the **adapter**. For example, if a device does DMA transfers, it must allocate **resources** to map the data buffer into the address space of the I/O bus. When a bus reset occurs, the bus-adapter routines revoke the allocation of all such resources and, consequently, a device driver must reallocate the resources it needs. If a transfer was in progress, the driver must then restart it. This routine is specific to the VAX architecture.

8.5 Autoconfiguration

Some hardware devices, such as the interface to the console terminal, are required for system operation. Other devices, however, may not be needed and their inclusion in the system may needlessly utilize system resources. Devices that might be present in different numbers, at different addresses, or in different combinations are difficult to configure in advance, however, and the system must support them if they are present and must fail gracefully if they are not present. To address these problems, 4.3BSD supports both a static *configuration procedure* that is done when a bootable system image is created, and a dynamic *autoconfiguration phase* that is done when the system is bootstrapped.

The static configuration procedure is done by the */etc/config* program. A configuration file is created by the system administrator that defines the set of hardware devices that might be present on a machine. This file identifies not only the types of devices, but also where each device might be located on the machine. For example, a system might be configured with two disk controllers and four disk drives that are connected in any of the configurations shown in Fig. 8.2. The configuration procedure generates several files that define the hardware topology. These files are compiled into the system for use in the autoconfiguration phase.

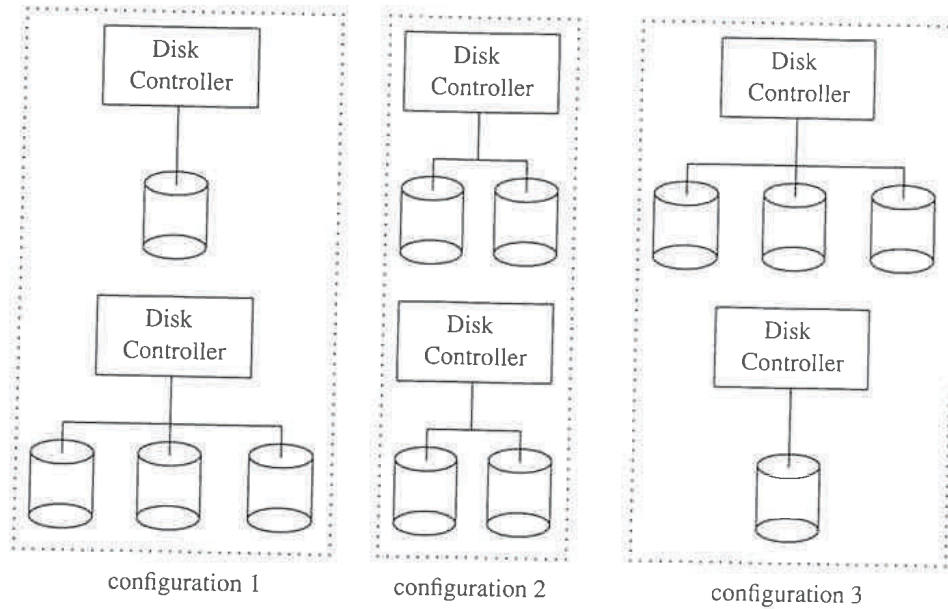


Figure 8.2 Drive configuration on the VAX.

The autoconfiguration phase is done during system initialization to identify the set of configured devices that are actually present on a machine. Autoconfiguration works by *probing* for configured devices at each of the possible locations where the device might be attached to the machine. Devices that are recognized during the autoconfiguration phase are *attached* and are made available for use. Devices that are present but not recognized remain unavailable until the system is rebooted.

Although this scheme requires that all the device drivers for hardware devices that might potentially be present on a machine be configured into a system, it permits device drivers to allocate system resources for only those devices that are present in a running system. It allows the physical device topology to be changed without requiring the system load image to be regenerated. It also prevents crashes resulting from attempts to access a nonexistent device. In the remainder of this section, we consider the autoconfiguration facilities from the point of view of the device-driver writer. We examine the device-driver support required to identify hardware devices that are present on a machine and the steps needed to *attach* a device once its presence has been identified. The internal structures of the configuration and autoconfiguration phases are left to Chapter 13.

Probing for Devices

During the autoconfiguration phase, a device-driver *probe routine* is called for each configured hardware device controller that responds to a memory access. The system passes to the probe routine the address of the controller's I/O registers

and expects the routine both to verify that the device is present and, if possible, to force the device to interrupt the host. If the probe routine is successful in forcing an interrupt, then the system will trap the interrupt and use the value of the vector to initialize the appropriate entries in the interrupt-vector table, so that the interrupt service routines for the device driver will be invoked on subsequent interrupts. For some hardware devices, it is impossible to force an interrupt reliably. In these instances, the system allows the probe routine to force a device to be configured by returning a known interrupt vector. If no interrupt is received and none is returned, the system assumes that the controller is not present at the supplied location.

In addition to probing for device controllers, a device driver may also be asked to probe for devices that may be attached to a controller. For example, the system will first probe to see whether a disk controller is present. For each controller found, the system will then probe for each possible disk drive that might be attached. Devices attached to a controller are termed *slave devices*. Disk drives and tape transports are the two types of slave devices. Device drivers that may have slave devices attached to their controller must provide a *slave routine* to probe for slave devices. The slave routine does not have to force an interrupt for each slave device; it needs only to return an indication of whether or not the slave device is present.

Attaching a Device

Once a device is found by a probe, the autoconfiguration code must *attach* it. Attaching a device is separated from probing to allow the system to initialize data structures used by the bus-adapter routines. Most device drivers use the attach routine to initialize the hardware device and any software state. For disk devices, for example, the attach routine is used to identify the geometry of the disk drive and to initialize the partition table that defines the placement of filesystems on the drive.

Device Naming

The autoconfiguration facilities support flexible placement of hardware on a machine by imposing a level of indirection in the naming of devices. Applications reference devices through block and character special files placed in the filesystem. The inode associated with a special file contains the major and minor device numbers of the associated hardware device. The major device number identifies the type of the device, whereas the minor device number identifies a *logical device unit*. For example, suppose the file `/dev/up1a` was created with the command

```
/etc/mknod /dev/up1a b 2 8
```

This file would refer to a block device with major device number 2 and minor device number of 8. Internally, the major device number would indicate a disk drive supported by the *up* device driver. The minor device number would be

passed to the device driver, where it would be interpreted according to the formula

$$\text{minor} = (8 \cdot \text{logical unit}) + \text{logical partition}$$

or, in this instance, partition 0 on logical unit (drive) 1. The *logical unit* for each device is assigned during the autoconfiguration phase and is distinct from hardware unit numbers used to identify devices. That is, whereas a tape unit or disk drive might have a hardware unit plug that identifies the device as physical unit x on a controller, to the system that device would be identified by a possibly different logical unit y . A logical unit may refer to different hardware devices each time a system is initialized or, more interesting, a specific hardware device may map to the same logical unit no matter where it is placed on the machine. This logical-to-physical mapping of device names within the system permits, for example, a disk drive to be shifted from one disk controller to another without rebuilding of the operating system. Flexibility in device naming is important in simplifying system maintenance in environments where redundant hardware is maintained for reliability. It also allows logical numbering to span controllers: There may be more than one hardware unit 0, whereas there can be only one logical unit 0.

8.6 UNIBUS Devices

The UNIBUS is an asynchronous bidirectional I/O bus that supports a wide variety of devices and moderate data-transfer rates; it is the original PDP-11 I/O bus. Each UNIBUS on a VAX is connected to the system bus through a *UNIBUS adapter*. Many devices may be attached to a UNIBUS. Several UNIBUS adapters may be attached to a VAX's system bus. The entire 18-bit address space of each UNIBUS is present in the VAX physical address space, and is mapped into the kernel's 32-bit virtual address space. This allows the kernel to access directly the whole address space on all UNIBUSes. The inverse mapping problem—providing a UNIBUS device access to VAX memory—is handled with a set of mapping registers in each UNIBUS adapter. These registers permit up to 248 Kbyte of VAX memory to be mapped into the address space of a UNIBUS at any one time. (The last 8 Kbyte of each UNIBUS address space, the *I/O page*, are reserved for device control register access.)

Many UNIBUS device controllers support DMA operations. In a DMA operation, the device controller moves data between main memory and the device without the assistance of the CPU. For a DMA operation to be done, the associated I/O buffer must be mapped into the address space of the UNIBUS using the mapping registers. In addition, a *data path* through the UNIBUS adapter must be allocated for use during the operation. Data paths act as a conduit for data transferred between a UNIBUS and the VAX's system bus. UNIBUS adapters support two types of data paths: *buffered* and *unbuffered*. The UNIBUS supports 8- and 16-bit data transfers, whereas the system bus on a VAX uses 32- or 64-bit transfers.

Buffered data paths collect the data from multiple UNIBUS transfers into single operations on the system bus, whereas unbuffered data paths do one system-bus transfer for each unit of data transferred on a UNIBUS. The use of a buffered data path reduces the number of system-bus transfers required during a DMA operation, which in turn reduces the load on the system bus and on the system as a whole. Each UNIBUS adapter provides a number of buffered data paths (from three to 15) that must be scheduled, along with the mapping registers, for each DMA transfer. Unbuffered data paths can be shared by devices, as they maintain no state.

In the next sections, we examine the structure and operation of device drivers for UNIBUS-based devices. The operation of the *up* device driver, a driver for an Emulex SC-21V disk controller, will be used as an example. Following our examination of the *up* device driver, we shall discuss the operation of the UNIBUS adapter-support routines.

The *up* Device Driver

The *up* device driver supports both raw and block-oriented access to multiple disk drives on the SC-21V controller. A variety of drives may be used with the controller; the driver automatically determines the identity and geometry of the drives during autoconfiguration. Aside from the more straightforward tasks of initiating and tracking I/O operations, the device driver also provides a number of reasonably sophisticated features:

- Overlapping seek operations with I/O operations
- Doing ECC correction on data transfers
- Automatically replacing sectors that have been marked as having uncorrectable media flaws with good sectors that are reserved as replacements

The *up* driver is described by entries in the block- and character-device tables. The *bdevsw* entry looks like the following:

```
{ upopen,      /* device open routine */
  nulldev,     /* device close routine */
  upstrategy,  /* i/o strategy routine */
  updump,      /* crash-dump routine */
  upsize       /* partition size routine */
},
```

The close entry point is not needed, so the default system null routine, *nulldev()*, is supplied in its place.

The *cdevsw* entry is similar:

```
{ upopen,      /* device open routine */
  nulldev,     /* device close routine */
  upread,      /* device read routine */
}
```



```

upwrite, /* device write routine */
nodev,   /* device ioctl routine */
nodev,   /* device stop routine */
upreset, /* device bus-reset routine */
seltrue, /* device select routine */
nodev    /* device mmap routine */
},

```

It makes use of the kernel default routines *nulldev()* (which returns a success indication) and *nodev()* (which returns an error indication).

Our look at the *up* driver is broken into the three areas described in Section 8.2: autoconfiguration support, routines for servicing I/O requests, and interrupt service routines. We shall concentrate initially on the block-oriented interface to the device, then shall return to consider the simple additions required to support a raw (character-device) interface.

Autoconfiguration Support

The *up* driver supports multiple disk controllers, each of which may have up to four disk drives attached. For the autoconfiguration phase, the driver must supply a *probe routine* to check for the presence of a disk controller, a *slave routine* to check for the presence of a disk drive on a controller, and an *attach routine* to attach each disk drive as that drive's presence is confirmed. To understand how these routines work, we must first understand a little about how the device itself functions.

The SC-21V controller is a DMA device that resides on the UNIBUS. The controller is manipulated through a bank of registers that is located in the UNIBUS I/O page and, through the autoconfiguration facilities, is made directly accessible to the device driver in the kernel's address space. The two most important registers on the controller, for the purposes of our discussion, are the *control status register* and the *drive status* register. The control status register contains controller-related status bits such as *controller ready* and is also used to select a specific drive on a controller when the driver is passing commands such as *start-read-operation* to the controller. The drive status register contains drive-related status information such as *drive on-line*. The content of the drive status register is meaningful only after the completion of a controller command. A no-operation command that completes immediately can be used by the host to poll for a drive's status.

To probe for a controller, the autoconfiguration routines test for the presence of a register at the beginning of the register bank. If a register is present, the autoconfiguration routines call the *uprobe()* routine, passing the address of a controller's register bank. The probe routine sets the controller ready with interrupts enabled and then pauses for a moment to allow the controller time to interrupt the CPU (see Fig. 8.3). After this delay, controller interrupts are disabled and the probe routine returns the size of the controller's register bank. If the controller is present, this sequence will cause it to interrupt the host. The system will

```

int upprobe(devaddr)
    device memory location *devaddr;
{
    force controller interrupt;
    delay 10 milliseconds to allow device interrupt;
    turn off controller interrupt;
    return (size of up register bank);
}

```

Figure 8.3 Algorithm for *up* driver controller probe.

trap the interrupt and will use the interrupt vector to install the *up* interrupt service routine, *upintr()*, in the appropriate entry in the interrupt-vector table. If the address at the beginning of the register bank does not happen to have anything plugged in, the UNIBUS adapter will note the error, and the autoconfiguration code will detect and handle it. If some other device happens to be at that address, the code that forces an interrupt is unlikely to make the other device interrupt.

Presuming that a controller is recognized, it will be assigned a logical unit number, and the bank of registers in the UNIBUS address space associated with the device will be marked as *in-use* (this is the reason *upprobe()* returns the size of its register bank). The autoconfiguration facilities keep track of the portions of the UNIBUS address space that have been allocated to devices to avoid allocation of overlapping regions by two devices.

To probe for a disk drive, the *upslave()* routine, shown in Fig. 8.4, is called. The drive to be checked is indicated by the value of *ui_slave*, a number specified at the time the system was configured. This value is given to the controller and the drive's status is requested from the controller. Depending on the controller's response, *upslave()* returns a value that indicates whether the disk drive was present.

The final routine associated with the autoconfiguration phase is the attach routine, *upattach()*, shown in Fig. 8.5. Attaching a drive involves some minor

Figure 8.4 Algorithm for *up* driver slave probe.

```

boolean upslave(ui, devaddr)
    unibus device *ui;
    device memory location *devaddr;
{
    select drive ui->ui_slave;
    request drive status;
    return (drive is present ? true : false);
}

```



```

upattach(ui)
    unibus device *ui;
    {
        set up mapping between unibus device and
            <controller, slave>;
        increment number of drives attached to this controller;
        call upmaptype to figure out type of disk drive;
    }

```

Figure 8.5 Algorithm for *up* driver attach.

bookkeeping tasks and one major task—figuring out the type of the drive. The *up* driver maintains a table that it uses to map <controller, slave> pairs to logical devices. This table is necessary because the *up* driver must be able to locate the data structures associated with a particular drive at interrupt time, given only the controller and slave numbers. In addition to setting up this mapping, the driver counts the number of drives attached to a controller. If two or more drives are found on a controller, then the driver will automatically overlap seek operations with I/O requests.

To deduce the type of disk drive attached to a controller, either the driver requests drive-geometry information from the controller, or, in the 4.3BSD Tahoe release, the driver tries to read a label from the disk with the geometry information in it. If the disk label is read correctly, no further information is needed from the controller. If the label cannot be read successfully, the geometry is deduced from a *personality PROM* on the controller, and is matched against a table of known geometries. If the geometry supplied by the controller matches a known drive's geometry, the drive presumes the associated drive type. Otherwise, a default drive type is supplied—most likely an incorrect one.

The drive type deduced in the attach routine is used to determine several important pieces of information either from the disk label, or, if no disk label exists, from tables compiled into the driver.

- The complete disk geometry: sectors per track, tracks per cylinder, and cylinders per drive. This information is needed to map a logical drive location into a cylinder, track, and sector for each I/O or positioning operation. (Some drives supply all this information directly, but no changes have been made to the general historical procedure for those drives.)
- The drive-partitioning information. Default drive partitions are defined on a per-drive-type basis because they start and end on cylinder boundaries for performance reasons.
- The seek optimization parameters. These parameters are dependent on a drive's physical characteristics; they are described later.

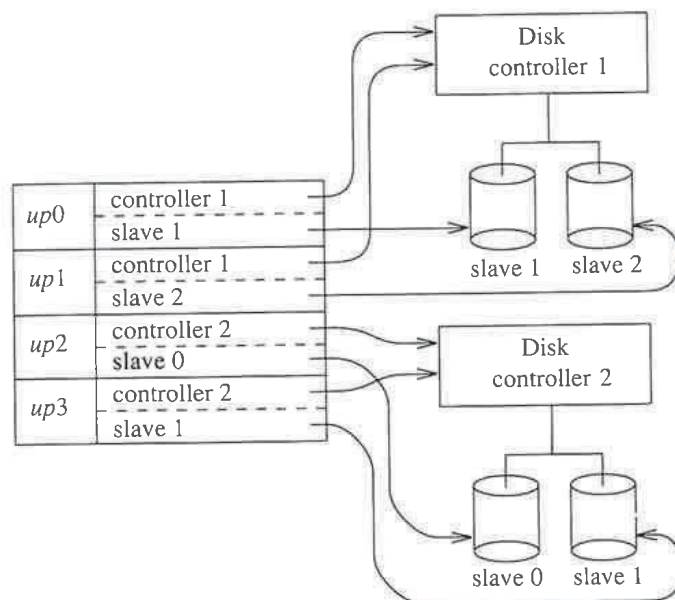
Historically, the drive-dependent information was compiled into the device driver. Recently, many vendors began using disk labels. The 4.3BSD Tahoe distribution has a comprehensive disk label that includes most of the parameters provided in the disk labels of the vendors. Disk labels have many advantages, not the least of which is that removable disk packs and fixed disk drives would be portable. That is, users could take media from one machine to another, or from just one type of controller to another, without having to depend on the software configuration for proper operation. Disk labels also allow each drive to use its own partition layout.

Logical-to-Device Mapping of I/O Requests

All I/O requests to the *up* driver are made through calls to the *upstrategy()* routine. For block-device access, each request to read or write data is encapsulated in a buffer structure allocated from the buffer pool. The block I/O system specifies requests in terms of logical blocks on a logical device. The *up* driver is responsible for mapping these *logical* units into units specific to the destination disk drive. The mapping of logical devices to physical hardware devices is done with data structures created during autoconfiguration. The minor device number specified in the buffer is used to index into a table of pointers to structures, each of which describe a disk drive. This data structure in turn contains a pointer to the structure describing the controller to which the drive is attached (see Fig. 8.6).

To map the logical disk block specified in an I/O request into a physical location on the disk drive, the driver consults partition tables associated with the drive.

Figure 8.6 Mapping of logical units to the physical device.



The partition tables define the base and extent of *logical drive partitions* in terms of sectors. The minor device encoding includes a drive partition. To calculate the physical location on the drive at which an I/O operation should commence, the driver simply adds the base offset of the indicated partition to the logical block specified in the buffer. The driver can further specify this location by using the drive-geometry information to calculate the cylinder, track, and sector of the start of the I/O transfer.

I/O Strategy

The *up* driver's strategy routine is passed I/O requests as they arise in normal system operation. The system does not order these requests in any way; it depends on the driver to stage them to the devices in an optimal fashion. Optimal scheduling of disk I/O traffic can be difficult. For most moving-head disk drives, the time to position a drive's read-write head at or near the location of an I/O transfer is normally much longer than the time it takes to transfer the data between the physical medium and memory in the controller. Consequently, the *up* driver strategy routine is mainly concerned with minimizing the time a controller spends waiting to position its read-write heads. When positioning time is minimized, one controller can most effectively be shared among multiple drives. An additional factor here is the implicit assumption that the disk controller is capable of overlapping positioning operations with data-transfer operations. The *up* driver and the Emulex SC-21V are typical: They allow multiple positioning operations to be carried out in parallel with a single transfer operation.

The staging of I/O requests is implemented with a collection of request queues, one per drive and one per controller. As requests come into the driver, they are sorted into the request queue for each drive by the *disksort()* routine, a systemwide utility routine that is used by most disk device drivers. The driver attempts to keep the controller as busy as possible by positioning drives at or near the location of a transfer in an operation separate from the actual transfer. That is, operations on each drive's request queue are first used to initiate a positioning operation on that drive; then, when the drive is positioned, the drive is moved onto the controller queue so that the I/O transfer can be done. Use of this strategy can be costly, however, if all I/O requests require two separate commands be given to the controller, because the completion of each command may require a separate interrupt to the host, and interrupt processing is a relatively costly operation. Consequently, the driver does not do positioning operations for all I/O requests. Instead, only those requests that are likely to allow additional concurrent requests are initiated.

The *upstrategy()* algorithm is shown in Fig. 8.7. The strategy routine utilizes two driver-specific routines, *upustart()* and *upstart()*, and the system's *disksort()* routine. The *upustart()* routine is responsible for deciding whether to initiate a positioning command for a drive and for adding the drive to the controller's request queue when the next I/O request is ready to start its transfer. The *upstart()* routine is responsible for initiating I/O transfers; it simply starts the

```

upstrategy(bp)
    buffer *bp;
{
    validate logical unit;
    check that transfer is within a drive's partition;
    calculate cylinder for transfer;
    raise priority to block upintr;
    call disksort to sort request into drive queue;
    if (drive is inactive) {
        call upstart to start drive;
        if (controller is inactive and drive is ready)
            call upstart to start controller;
    }
    lower priority;
}

```

Figure 8.7 Algorithm for *up* driver I/O strategy.

transfer operation specified by the request for the drive on the front of the controller's queue.

Disksort

The *disksort()* routine is called by disk drivers to sort I/O requests into a drive's request queue using an *elevator sorting algorithm*. This algorithm sorts requests in a cyclic, ascending, cylinder order, so that requests can be serviced with a minimal number of one-way scans over the drive. This ordering was originally designed to support the normal read-ahead requested by the filesystem as well as to counteract the filesystem's random placement of data on a drive. With the improved placement algorithms in the current filesystem, the effect of the *disk-sort()* routine is less noticeable; it produces the largest effect when there are multiple simultaneous users of a drive.

The *disksort()* algorithm is shown in Fig. 8.8. A drive's request queue is made up of one or two lists of requests ordered by cylinder number. The request at the front of the first list indicates the current position of the drive. If a second list is present, it is made up of requests that lie before the current position. Each new request is sorted into either the first or second list according to the request's location. When the heads reach the end of the first list, the drive begins servicing the other list.

Disksort can also be important on machines that have a fast processor but that do not sort requests within the device driver. In this situation, if a write of several Kbyte is honored in order of queueing, it can block other processes from accessing the disk while it completes. Sorting requests provides some scheduling, which more fairly distributes accesses to the disk controller.


```

disksort(dq, bp)
    drive queue *dq;
    buffer *bp;
{
    if (drive queue is empty) {
        place the buffer at the front of the drive queue;
        return;
    }
    if (request lies before the first active request) {
        locate the beginning of the second request list;
        sort bp into the second request list;
    } else
        sort bp into the current request list;
}

```

Figure 8.8 Algorithm for disk sort.

Drive-Positioning Algorithm

The algorithm used by the *upstart()* routine is shown in Fig. 8.9. The basic idea is clear: examine the next request on the drive's request queue. If the drive is not positioned near the location of the request, then initiate a positioning operation. Otherwise, the drive is positioned properly, so place the drive on the controller's queue so that the I/O operation will be done. The only interesting part of the algorithm to consider is the heuristic used in deciding when a drive should be positioned.

The *up* driver maintains two parameters, termed *sdist* and *rdist*, for each type of drive. These parameters are thresholds applied to the rotational distance from the current drive position to the position of the next I/O transfer. If the current position is in the same cylinder as the next transfer and is between *sdist* and *sdist + rdist* rotational distances, then no explicit positioning command is requested; instead, positioning is left to be done by the controller at the time the transfer takes place. Otherwise, a *search command* is issued to move the heads to the correct cylinder and then to wait for the sector at a distance of *sdist* from the desired sector.

The *sdist* and *rdist* parameters are chosen according to properties of the disk drive and controller. The goal is to ensure that the controller can complete the next I/O operation before the drive rotates the data past the position of the read-write heads. If the controller is told to initiate an I/O transfer after the starting point of the requested data has passed under the read-write heads, then the controller will be forced to wait for the data to rotate around until they are again under the read-write heads. This condition can be costly to performance, as the time spent waiting for the drive to revolve might have been used for other transfers. The *up* driver's heuristic attempts to ensure that this condition will not occur.

```

upstart(ui)
    unibus device *ui;
{
    if (no request on drive queue) return;
    if (controller is busy) {
        mark device as needing attention when controller is
        idle;
        return;
    }
    if (drive is not positioned over cylinder) {
        mark drive positioned over cylinder;
        if (controller has more than one drive &&
            heads are far from transfer point) {
            select drive;
            position drive at or near location of transfer;
            return;
        }
    }
    place drive on controller queue;
}

```

Figure 8.9 Algorithm for *up* driver unit start.

Many newer disk controllers provide the scheduling optimizations that the *up* driver provides internally. Having the controller make these calculations is a benefit for operating systems that do not directly support these facilities. However, since a disk controller cannot have all the information available to an operating system, there can be instances of anomalous behavior. For example, the controller cannot give synchronous requests a precedence higher than that it gives asynchronous ones.

Initiating an I/O Operation

When the driver decides that an I/O operation should be started, it invokes the *upstart()* routine, as shown in Fig. 8.10. The task of *upstart()* is straightforward: examine the next request on the controller queue and tell the controller to start the I/O operation. On machines where device controllers are directly attached to the main system bus, this is perhaps all that is needed. For the VAX, however, the device driver must call the UNIBUS adapter-resource routines to request resources needed in doing the transfer. The system allocates these resources dynamically, rather than statically at boot time, because many devices on a single bus adapter may together require more resources than are available in an adapter. Furthermore, certain I/O devices require exclusive use of the bus when doing data transfers because the device controller has insufficient data buffering. For these reasons, to initiate a DMA I/O operation, a device driver first sets up all the


```

upstart(ui)
    unibus device *ui;
{
    examine first request on controller queue;
    if (no request to process)
        return;
    mark controller busy and determine destination of request;
    select drive and check that it is ready and on-line;
    set up I/O transfer;
    call ubago to schedule adapter resources for transfer;
}

```

Figure 8.10 Algorithm for *up* driver controller start.

device-specific parameters for a transfer and then calls the *ubago()* routine to allocate resources in the UNIBUS adapter. When resources are available, the *ubago()* routine then calls back to the device driver to begin the I/O transfer. Once a transfer has been started, the work of the driver's strategy routine is complete. All that is left is for the controller to do the transfer and to deliver an interrupt to the host when the transfer is completed. Notice that queueing delays forced by the adapter routines can adversely affect the performance of a system. For this reason, tuning the performance of a VAX system requires careful consideration of the load that I/O devices will place on a UNIBUS.

Interrupt Handling

The *up* driver has a single interrupt service routine, *upintr()*. Interrupts are posted by the controller as each I/O or positioning operation completes. The interrupt-handling algorithm is shown in Fig. 8.11.

Interrupts may arrive for either I/O or positioning operations, but positioning interrupts that occur during a transfer operation will be deferred until the transfer operation is completed. So *upintr()* must check whether an I/O operation is pending, and if one is, must check the operation's completion status. Then, it scans for drives that have completed search operations. The *upstart()* routine is called to schedule the next operation on each drive that has completed its current request.

When an I/O request has been completed, *upintr()* notifies the process that requested the I/O operation by calling the *biodone()* routine. This routine wakes up the blocked process, frees the buffer for an asynchronous write, or, for a request by the *pagedaemon* process, invokes a procedure specified in the buffer structure.

Aside from the handling of successful I/O operations, the interrupt service routine is also responsible for handling the many types of errors that can occur. Although certain errors are unrecoverable, others may be handled by the device driver. Two errors in particular that the device driver will attempt to correct are requests that have a short error burst in the data transferred, and requests for

```

upintr(cntrl)
    controller *cntrl;
{
    if (controller was doing a transfer) {
        get buffer associated with transfer;
        select drive and check drive and controller status;
        process errors on either drive or controller;
        if (operation completed) {
            take request off controller queue;
            record transfer status;
            call biodone to notify I/O is complete;
            if (this drive has more operations pending)
                call upustart to start next
                    operation on drive;
        }
        release unibus resources used for transfer;
    }
    for (each drive with a completed seek)
        call upustart to start next operation on drive;
    if (devices are ready to transfer)
        call upstart to start next operation on controller;
}

```

Figure 8.11 Algorithm for *up* driver interrupt handling.

which the data have been placed in an alternate location on the drive. Recovery from the first type of error is done with an error-correction code (ECC) supplied by the controller to correct the data transferred by the controller. Recovery from the second type of error requires that the device driver look up the *bad sector* in a *bad-sector table* to locate the alternate sector. If the faulty sector is found in the bad-sector table, the driver initiates a transfer from the alternate location and returns those data as though they had been at the location originally requested.

The two error-recovery mechanisms described here are commonly termed *ECC correction* and *bad-sector forwarding*. Most current disk controllers provide both of these recovery schemes built into the controller. Bad-sector forwarding can be implemented in several ways. One scheme reserves a single region on a disk for mapping sectors with uncorrectable flaws. This region typically consists of several cylinders near the end of the drive. The problem with this method is that, when a bad sector is identified, the alternate sector may be located far away from the current position. Thus, accessing the alternate sector is a time-consuming positioning operation. To address this problem, a second scheme, termed *slip-sector forwarding*, is often used. Instead of one contiguous region being reserved, one sector on each track is reserved. In this scheme, an alternate *slip sector* can be used to replace a bad sector with at most a delay of a single disk revolution to reach the alternate sector. However, if more than one sector goes

bad on a track, there is no way to remap the second sector that fails. Manufacturers commonly study the statistical failure properties of disk-drive media to determine, for example, how many slip sectors to allocate on a track and how many cylinders to reserve on a drive. Hybrid schemes that incorporate both approaches have been used as well; these tend to be the most effective, but can be costly to implement entirely within a controller. In general, bad-sector handling can have a noticeable effect on performance, but is preferable to discarding equipment (especially as current disk drives seldom have flawless media).

Adding raw access to the *up* driver is simple. All that is needed are two routines, *upread()* and *upwrite()*, and a set of buffers for doing raw I/O. The read and write routines simply invoke the *physio()* routine as described in Section 8.4.

UNIBUS Adapter Support Routines

The UNIBUS adapter routines provide two major functions in supporting drivers written for UNIBUS devices:

1. They manage each adapter's resources, dynamically allocating portions to drivers so the drivers can do DMA transfers.
2. They watch for adapter failures, resetting the adapter's state when the adapter reports a serious or recurrent error.

(The second function was required by the initial instability of UNIBUS adapters; this function is needed less frequently today.)

Each UNIBUS adapter has a set of registers that is used to map pages of memory into the address space of the UNIBUS. There are 496 such registers, each mapping one 512-byte page, enough to map 248 Kbyte of memory. The remaining 8 Kbyte of each UNIBUS address space is reserved for UNIBUS device registers. For a DMA transfer between main memory and a UNIBUS peripheral to be done, the I/O buffer must be mapped into the UNIBUS address space with a contiguous set of mapping registers, and a *buffered* or *unbuffered* data path must be allocated for use during the transfer. Buffered data paths cannot be shared by multiple devices doing DMA; thus, their use must be scheduled on a per-transfer basis.

I/O operations through a UNIBUS adapter are done in three steps:

1. Allocate resources for an operation
2. Initiate the operation
3. On notification that the operation has completed, release the resources allocated in step 1

The work of steps 1 and 3 is normally hidden inside the *ubago()* and *ubadone()* routines, respectively. The work of step 2 is device-specific and is carried out by a device driver's *go routine*, which is automatically invoked when resources have been allocated for an operation.

```

boolean ubago(ui)
    unibus device *ui;
{
    locate unibus adapter for device;
    raise priority to spluba;
    if (device needs exclusive use of adapter and other
        users exist || adapter is exclusively in use)
        goto resource_wait;
    call ubasetup to allocate unibus adapter resources for
        transfer (indicate no wait is to be done in the
            routine);
    if (not all resources were available)
        goto resource_wait;
    increment number of users of adapter;
    if (device needs exclusive use of adapter)
        mark adapter as exclusively in use;
    lower priority;
    if (this device was previously on adapter's waiting queue)
        remove device entry from adapter queue;
    call device's go routine to start operation;
    return (true);

resource_wait:
    place device on queue of devices awaiting adapter
        resources;
    lower priority;
    return (false);
}

```

Figure 8.12 Algorithm for scheduling a UNIBUS I/O operation.

The algorithm used by *ubago()* is shown in Fig. 8.12. Device drivers allocate resources implicitly through the call to the *ubasetup()* routine (the algorithm used by *ubasetup()* is shown in Fig. 8.13). If a device requires exclusive use of an adapter, or the adapter is currently in use by a device that requires exclusive access, then the device is placed on a queue of waiting devices maintained for each adapter. Similarly, if resources are currently unavailable, the device is placed on the adapter's queue.

When a driver receives an interrupt and releases resource through a call to *ubadone()* (see Fig. 8.14), the adapter's queue is scanned. Each device that is present on the queue is given another chance to allocate resources and to initiate its I/O operation. Processes that are blocked inside of *ubasetup()* awaiting resources are awakened to try again.

There is some subtlety in the implementation of these algorithms. The adapter routines must take care to guard against a bus reset that is caused by an


```

boolean ubasetup(number, bp, flags)
    unibus adapter number;
    buffer *bp;
    int flags;
{
    calculate number of adapter mapping registers needed for
        transfer;
    raise priority to spluba;
    while (mapping registers are unavailable) {
        if (flags indicate caller cannot wait) {
            lower priority;
            return (false);
        }
        increment count of waiters for mapping registers;
        block to await mapping registers;
    }
    allocate mapping registers required;
    if (a buffered data path is required) {
        while (no buffered data path is available) {
            if (flags indicate caller cannot wait) {
                release allocated
                    mapping registers;
                lower priority;
                return (false);
            }
            increment count of waiters for
                buffered data paths;
            block to await a buffered data path;
        }
        allocate a buffered data path;
    }
    lower priority;
    record allocated adapter resources;
    calculate physical location of data buffer;
    set up adapter mapping registers to point to data buffer;
    return (true);
}

```

Figure 8.13 Algorithm to allocate UNIBUS adapter resources.

unrecoverable adapter error or by a timeout on an I/O operation. When a reset occurs, any resources currently allocated for an I/O operation are reclaimed directly by the UNIBUS adapter reset routine. Thus, *ubarelse()* (see Fig. 8.15) must take care when freeing resources so that resources are not deallocated twice. To avoid this problem, *ubarelse()* raises the processor priority to *spluba* while it

```

ubadone(um)
    unibus controller *um;
{
    locate unibus adapter for controller;
    if (device required exclusive use of adapter)
        mark adapter as no longer in exclusive use;
    decrement number of users of adapter;
    call ubarelse to release resources used in transfer;
}

```

Figure 8.14 Algorithm for completing a UNIBUS I/O operation.

is releasing resources so it can block the bus-adapter reset routine that is invoked as the result of a UNIBUS adapter interrupt. Because the priority level during deallocation is raised, *ubarelse()* can do a consistency check on the resources it is freeing and can free them before the resources may be deallocated by the bus reset routine.

Figure 8.15 Algorithm for releasing UNIBUS adapter resources.

```

ubarelse(number, mapptr)
    unibus adapter number;
    mapping register pointer *mapptr;
{
    raise priority to spluba;
    if (buffered data path was used) {
        free data path;
        if (there are waiters for buffered data paths) {
            zero the count of waiters;
            wake up all processes waiting for a
                buffered data path;
        }
    }
    deallocate mapping registers used;
    lower priority;
    if (there are waiters for mapping registers) {
        zero the count of waiters;
        wake up all processes waiting for mapping registers;
    }
    while (there are requests queued for the adapter) {
        if (call ubago indicates operation could not be done)
            break;
    }
}

```


8.7 MASSBUS Devices

DEC designed the MASSBUS as an I/O bus for higher-speed mass-storage peripherals than could be supported by the UNIBUS. The MASSBUS has a greater bandwidth than does the UNIBUS, but it also has a more rigid structure. A MASSBUS can support at most eight devices, with only one at a time transferring data. The only devices available for a MASSBUS are disks and tapes. Tape formatters can each support up to eight tape transports. Each MASSBUS on a VAX is connected to the system bus via a *MASSBUS adapter*. Since the MASSBUS design supports higher transfer rates than those the UNIBUS supports, some vendors have designed device controllers that connect directly to the system bus and emulate a MASSBUS adapter.

The rigid structure of the MASSBUS permits a significant portion of each MASSBUS device driver to be isolated in a common set of *MASSBUS adapter-support routines*. These routines manage most of the I/O request queueing and interpose their own interrupt service routines between the devices and the device drivers. MASSBUS device drivers differ from UNIBUS device drivers in the way they schedule adapter resources for each DMA operation. The complicated resource scheduling required for a UNIBUS adapter is not needed, because the MASSBUS allows only one device to transfer data at a time. The use of a single data path simplifies I/O scheduling but can potentially add queueing delays.

Autoconfiguration

MASSBUS devices are probed and attached in a fashion similar to that used for UNIBUS devices. The major difference is that MASSBUS devices can be probed in a device-independent fashion because certain controller status registers in a MASSBUS adapter are defined with standard bit assignments. This means that a single MASSBUS probe routine can be written that identifies each of the eight possible devices on a MASSBUS adapter. Only the identification of tape transports must be handled on a per-device basis. Thus, autoconfiguration support in a MASSBUS device driver is reduced to an attach routine and, if the device is a tape controller, to a slave routine.

I/O Strategy

The only MASSBUS devices are tape and disk controllers. As these devices provide only raw and block-oriented interfaces, all I/O requests eventually pass through a device driver's strategy routine. Unlike scheduling in the *up* device driver, the I/O scheduling for MASSBUS devices is done mostly in the MASSBUS adapter-support routines. For example, the *htstrategy()* routine for the *ht* magnetic-tape device, shown in Fig. 8.16, is simple: It just places each request at the end of the transport's request queue and, if the transport is not currently active, calls the MASSBUS adapter-support routine *mbustart()* to start activity on the unit.

The I/O request scheduling algorithm is structured much like the algorithm used in the *up* driver. A MASSBUS adapter unit start routine, *mbustart()*, is called

```

htstrategy(bp)
    buffer *bp;
{
    raise priority to splbio;
    if (request queue for this tape transport is empty)
        place bp at the front of the queue;
    else
        place bp at the end of the queue;
    if (tape transport is not active)
        call mbustart to start an operation on transport;
    lower priority;
}

```

Figure 8.16 Algorithm for *ht* driver strategy.

by a device driver after one or more operations have been placed on the device's request queue. The *mbustart()* routine takes the operation at the head of the device's request queue and *calls back* to the device driver's *unit start routine* to initiate the operation. The unit start routine carries out whatever actions are needed for the specified request and then returns an indication that one of the following occurred:

- The request completed immediately (e.g., a *sense* operation on a tape drive).
- The request could not be completed because the device was busy (e.g., a dual-ported disk drive was servicing its other port at the time of the request).
- The request was started (for a non-data transfer operation).
- The request involved a data transfer and the device should be scheduled for access to the single data path on the MASSBUS adapter.

In the case of a data transfer, the device is placed on the MASSBUS adapter's data-transfer request queue and, if the adapter's data path is idle, the unit start routine starts the transfer immediately by calling the *mbstart()* routine.

The *mbstart()* routine is functionally similar to the *ubago()* routine described in the previous section. Unlike operations on UNIBUS adapters, however, the I/O operations are initiated in the *mbstart()* routine rather than in the device-driver start routine. This is possible because the register banks for all MASSBUS devices have a command register at a known location.

To begin a transfer, the driver's start routine is first invoked to fill in drive-specific registers and to determine the value to be placed in the device's command register. Then the data buffer is mapped into the MASSBUS address space so that the device can access the associated data buffer, and finally the command supplied by the driver's start routine is placed in the command register to initiate the operation.

Interrupt Handling

Interrupts by MASSBUS devices generate a single MASSBUS adapter interrupt to the CPU. The MASSBUS adapter interrupt service routine is invoked with the


```

mbintr(number)
    massbus adapter number;
{
    retrieve device status from massbus adapter;
    if (adapter was active doing a data transfer) {
        identify device that was transferring data;
        call device's data transfer complete interrupt
            routine;
        switch (status of data transfer) {
        case transfer done:
            remove request from device request queue;
            notify process that initiated operation;
            /* fall through... */
        case retry transfer because of error:
            remove request from adapter request queue;
            mark adapter not active doing data transfer;
            if (device's request queue is not empty)
                call mbustart to start device's next
                    operation;
            break;
        }
    }
    for (each device needing attention) {
        if (device has nondata transfer interrupt routine) {
            call device's nondata transfer interrupt routine;
            switch (status of nondata transfer) {
            case operation completed:
                remove request from device request queue;
                notify process that initiated operation;
                /* fall through... */
            case retry operation because of error:
                if (device's request queue is not empty)
                    call mbustart to start device's next
                        operation;
                break;
            }
        } else {
            call mbustart to start device's next operation;
        }
    }
    if ((the adapter request queue is not empty) &&
        (adapter is not marked as doing a data transfer))
        call mbstart to initiate next I/O operation;
}

```

Figure 8.17 Algorithm for MASSBUS adapter interrupt handling.

identity of the interrupting MASSBUS adapter, and must read a status register in the adapter to find out which devices require service. The interrupt-processing algorithm for a MASSBUS adapter is shown in Fig. 8.17. Each device driver may supply two interrupt service routines, one for operations that involve data transfer and one for operations that do not involve data transfer. Both types of routines must return a status code that indicates whether an operation has completed or needs to be retried because of an error (additional status codes exist that are not important to our discussion). The MASSBUS interrupt service routine uses these status codes to drive a *finite-state machine* that controls the scheduling of operations through the adapter.

Exercises

- 8.1 Would you expect a cartridge tape drive to have a block or character interface? Why?
- 8.2 Why do filesystem utilities such as `fsck` usually use the character-device interface to access disk drives?
- 8.3 Why are interrupt service routines **running** in the bottom half of the kernel precluded from calling the `sleep()` routine?
- 8.4 What responsibilities do device probe routines have during the auto-configuration phase of system startup?
- 8.5 Explain the purpose and operation of the glue routines that 4.3BSD installs in the interrupt-vector table.
- 8.6 Why are two lists used by the `disksort()` routine in ordering I/O requests for a disk drive?
- 8.7 Adapter queueing of I/O requests **destined for a UNIBUS or MASSBUS device** can ruin disk-drive scheduling **done by the device driver**. What are the tradeoffs between doing seek optimization in device drivers and doing seek optimization in a disk controller?
- 8.8 Most hardware devices permit only one I/O operation at a time. How is this limit enforced by 4.3BSD?
- 8.9 Suppose you wanted to **modify** the `up` device driver for a disk controller that **handles multiple simultaneous I/O requests**. What assumptions in the I/O queueing structure of the driver would you have to change?
- 8.10 The device `open()` routine is called for each `open` system call, but the device `close()` routine is called on only the last close on a device. Why were these semantics chosen? Give an example of a device where these semantics cause problems.
- 8.11 What function does a UNIBUS adapter provide to support the connection of a UNIBUS on a VAX?

- 8.12 The *uprobe()* routine, shown in Fig. 8.3, waits for 10 milliseconds to allow the device to interrupt. How could you implement this delay? How would the CPU's performance affect the implementation that you have selected?
- 8.13 Why does 4.3BSD interpret the minor device number as a logical rather than as a physical device unit number?
- 8.14 Describe what a disk driver must do to support overlapped seeks. What is ECC correction? What is bad-sector forwarding? What is a slip sector?
- 8.15 Explain how the *sdist* and *rdist* parameters are used in the *up* disk driver.
- 8.16 Why does 4.3BSD require all UNIBUS device drivers to stage I/O operations through the UNIBUS adapter routines?
- 8.17 Why are the MASSBUS adapter routines able to do more work than are the UNIBUS adapter routines?
- 8.18 In the *physio()* routine shown in Fig. 8.1, the processor priority is raised to *splbio* before the content of the buffer header is used. Why is it necessary to raise the processor priority level?
- *8.19 In the *physio()* routine shown in Fig. 8.1, the processor priority is raised and lowered several times. Could you instead raise the level once on entry and lower it on exit? Explain your answer.
- *8.20 The disk label, containing the drive geometry and logical partitions, is read in by the disk driver when the first partition on a disk is opened. Where must the disk label be located? Why must it be in that place?
- *8.21 Some operating systems allow device drivers to be loaded into a running system. What are the tradeoffs of this scheme?
- *8.22 Suppose a device did not use vectored interrupts, but instead asserted the interrupt signal until it was told to stop. What implications would this have for the way 4.3BSD handles interrupts on the VAX? How would this affect the autoconfiguration scheme?
- *8.23 Describe the interaction between the algorithms used by the filesystem to place data on a disk drive and the I/O scheduling algorithms embodied in the *disksort()* routine.
- **8.24 Construct a model for disk-drive scheduling that you can use to calculate accurate values for the *sdist* and *rdist* parameters used by the *up* driver. How does the speed of the CPU affect your model?
- **8.25 Describe an alternative scheme for managing UNIBUS resources during a bus reset.
- **8.26 Consider a shared-memory multiprocessor architecture in which all devices are accessible on all processors. What changes to the I/O system and device drivers would be required to permit I/O operations to be managed by any processor?

Terminal Handling

One of the most common types of hardware peripheral device found on UNIX systems is an interface supporting one or more terminals. The most common type of interface is a *terminal multiplexer*, a device that connects multiple asynchronous RS-232 serial lines, which may be used to connect terminals, modems, printers, and similar devices. Unlike the block storage devices described in Chapter 8 and the network devices to be considered in Chapter 11, terminal devices often process data one character at a time. Like other devices, terminal multiplexers are controlled by device drivers specific to the actual hardware.

Terminal interfaces interrupt the processor asynchronously to present input, which is independent of process requests to read user input. Data are processed when they are received, and then are stored until a process requests them, thus allowing *type-ahead*. Most terminal ports are used to attach local or remote terminals on which users may log in to the system. When used in this way, terminal input represents the keystrokes of users, and terminal output is printed on the users' screens or printers. We shall deal mostly with this type of terminal line usage in this chapter. Asynchronous serial lines are also used to connect modems for computer-to-computer communications or serial-interface printers. When serial interfaces are used for these purposes, they generally use a subset of the system's terminal-handling capability. In some cases, they use special processing modules for higher efficiency. We shall discuss alternate terminal modules later in this chapter.

9.1 Terminal Processing Modes

4.3BSD UNIX supports several modes of terminal processing. Much of the time, terminals are in *line mode* or *cooked mode*, in which input characters are echoed by the system as they are typed, and are collected until a carriage return is

received. Only after the receipt of a carriage return is the entire line made available to the shell or other process reading from the terminal line. In line mode, the user may correct typing errors, deleting the most recently typed character with the *erase character*, deleting the last word with the *word-erase character*, or deleting the entire current line with the *kill character*. Other special characters generate signals sent to the current processes associated with the terminal; these signals may abort processing or may suspend it. Additional characters start and stop output, flush output, or prevent special interpretation of the succeeding character. The user can type several lines of input, up to an implementation-defined limit, without waiting for input to be read and then removed from the input queue. The user can select the special processing characters or even selectively disable them.

Screen editors and programs that communicate with other computers generally run in *character-at-a-time mode*. The system supports two such modes. In *cbreak mode*, the system makes each typed character available to be read as input as soon as that character is received, and thus erase and kill processing are not done. Other special characters are processed normally, allowing signals and flow control as in line mode. The other character-at-a-time mode is *raw mode*, in which all special character input processing is disabled; all characters are passed to the program reading from the terminal.

In addition to processing input characters, terminal interface drivers must do certain processing on output. Most of the time, this processing is quite simple. UNIX newlines are converted to a carriage return plus a line feed, and the interface hardware is programmed to generate appropriate parity bits on output characters. Some terminals require extra time to process characters that produce motion such as carriage return, newline, and tab; thus, the terminal driver may be set to insert a delay after each such character. In addition to character processing, the terminal output routines must manage flow control, both with the user (using stop and start characters) and with the process. As terminal devices are quite slow in comparison with other computer peripherals, a program writing to the terminal may produce output much faster than that output can be sent to the terminal. The process must thus be blocked when it has filled the terminal output queue, and must be restarted when sufficient output has drained.

9.2 Line Disciplines

Most of the character processing done for terminal interfaces is independent of the type of hardware device used to connect the terminals to the computer. Therefore, most of this processing is done by common routines in the *tty driver* or terminal handler. Each hardware interface type is supported by a specific device driver. The hardware driver is a device driver like those described in the previous chapter; it is responsible for programming the hardware multiplexer. It is responsible for receiving and transmitting characters and for handling some of the synchronization with the process doing output. The hardware driver is called by the *tty driver* to do output; in turn, it calls the *tty driver* with input characters as they

Table 9.1 Entry points of a line discipline.

Routine	Called from	Usage
<i>l_open</i>	above	initial entry to discipline
<i>l_close</i>	above	exit from discipline
<i>l_read</i>	above	read from line
<i>l_write</i>	above	write to line
<i>l_ioctl</i>	above	control operations
<i>l_rint</i>	below	received character
<i>l_rend</i>	none	unused
<i>l_meta</i>	none	unused
<i>l_start</i>	below	completion of transmission
<i>l_modem</i>	below	modem carrier transition

are received. Because serial lines may be used for more than just connection of terminals, a modular interface between the hardware driver and the tty driver allows either part to be replaced with alternate versions. The tty driver interfaces with the rest of the system as a *line discipline*. A line discipline is a processing module used to provide semantics on an asynchronous serial interface (or, as we shall see, on a software emulation of such an interface). It is described by a procedural interface, the *linesw* (line switch) structure.

The *linesw* structure specifies the entry points of a line discipline, much as the character-device switch *cdevsw* lists the entry points of a character device driver. Like all device drivers, a terminal multiplexer driver is divided into the top half, which runs synchronously when called to process a system call, and the bottom half, which runs asynchronously when device interrupts occur. The line discipline provides routines that do common terminal processing for both the top and bottom halves of a terminal hardware driver. The entry points of a line discipline are listed in Table 9.1.

Device drivers for serial terminal interfaces support the normal set of character-device-driver entry points specified by the character-device switch. Several of the standard driver entry points (*read*, *write*, and *ioctl*) immediately transfer control to the line discipline when called. (The standard tty select routine *ttselect()* usually is used as the device driver *select* entry in the character-device switch.) The *open* and *close* routines are similar; the line-discipline open entry is called when a line first enters a discipline, either at initial open of the line or when the discipline is changed. Similarly, the discipline close routine is called to exit from a discipline. All these routines are called from above, in response to a corresponding system call. The remaining line-discipline entries are called by the bottom half of the device driver to report input or status changes detected at interrupt time. The *l_rint* (receiver interrupt) entry is called with each character received on a line. The corresponding entry for transmit-complete interrupts is

the *l_start* routine, which is called when output operations complete. This entry gives the line discipline a chance to start additional output operations. For the normal terminal line discipline, this routine simply calls the driver's output routine to start the next block of output. Transitions in modem control lines (see Section 9.7) may be detected by the hardware driver, in which case the *l_modem* routine is called with an indication of the new state.

The system includes several different types of line disciplines. Most lines use one of the two terminal-oriented disciplines described in the next section. Other disciplines in the system support graphics tablets on serial lines and asynchronous serial network interfaces.

9.3 User Interface

The terminal line discipline used by default on most terminal lines is derived from the discipline present in UNIX 32V and Seventh Edition UNIX. The user interface to this line discipline, known as the *old terminal driver*, is still present in 4.3BSD. An extension of the old driver was added to the system beginning with 4.1BSD to support new signals for job control and to provide logical line-editing facilities on CRT terminals. The extended driver, called the *new terminal driver*, retains the interface of the old driver but adds new options and modes to the previous set. Several of the options added in the new driver modify basic modes of the previously existing driver, thus creating a rather cluttered interface. Both new and old terminal line disciplines are implemented by the same terminal driver, which changes its behavior according to the current discipline.

The main user interface for control of the terminal line discipline is the *ioctl* system call. This call is used to change disciplines, to set and get values for special processing characters and modes, to set and get hardware serial line parameters and to do other control operations. Most *ioctl* operations require one argument in addition to a file descriptor and the command; the argument is the address of an integer or structure from which the system gets parameters, or into which return information is placed. The following set of *ioctl* commands apply specifically to the standard terminal line disciplines, although all line disciplines must support at least the first two. Other disciplines generally support other *ioctl* commands.

TIOCGTD	Get the identity of the line discipline for this line as an integer constant
TIOCSETD	Set the discipline for this line
TIOCGTEP	Get the basic (old terminal driver) parameters for this line, including line speed, basic modes, and erase and kill characters
TIOCSETP	Set the basic parameters for this line
TIOCSETN	Set the basic parameters for this line without flushing input

TIOCGETC	Get values of the basic set of special characters (other than erase and kill)
TIOCSETC	Set the values of the special characters supported by the old terminal driver (other than erase and kill)
TIOCLGET	Get (set, add, delete) the <i>local</i> options—those used in only the new terminal driver
TIOCLSET	
TIOCLBIS	
TIOCLBIC	
TIOCGLTC	Get (set) the <i>local</i> special characters—those used by only the new terminal driver
TIOCSLTC	
TIOCGWINSZ	Get (set) the terminal or window size for the terminal line; the window size includes width and height in characters and (optionally, on graphical displays) in pixels
TIOCSWINSZ	

This set of terminal *ioctl* commands illustrates the gradual growth of the set of options supported by the terminal interface.

The parameters set by TIOCSETP and TIOCLSET include flags that modify the behavior of the terminal line. In particular, two flags (RAW and CBREAK) control the overall mode of the *tty*, and others control additional features or modify the overall mode. The RAW flag sets the terminal in *raw* mode; the CBREAK flag sets the terminal in *cbreak* mode; with neither flag set, the terminal is in *cooked* mode, the normal, line-at-a-time mode.

9.4 The *tty* Structure

Each terminal hardware driver contains a data structure to contain the state of each line that it supports. This structure, the *tty structure*, contains state information, the input and output queues, the modes and options set by the *ioctl* operations listed previously, and the line discipline number (see Table 9.2). The *tty structure* is shared by the hardware driver and the line discipline. The calls to the line discipline all require a *tty structure* as a parameter; the driver locates the correct *tty* according to the minor device number. This structure also contains information about the device driver needed by the line discipline.

The sections of the *tty structure* include

- State information about the hardware terminal line. The *t_state* field includes line state (open, carrier present, or waiting for carrier) and major file options

Table 9.2 The *tty* structure.

Type	Description
Character queues	Raw input queue
	Canonical input queue
	Device output queue
Hardware parameters	Device number
	Output procedure
	Terminal column number
Selecting	Process selecting for reading
	Process selecting for writing
State	Process group
	Flags
	Line discipline
	Number of rows and columns
	Input and output baud rates
	Special characters on input

(use of nonblocking I/O or signal-driven I/O). Transient state for flow control, delay processing, and synchronization also is stored here.

- Input and output queues. The hardware driver transmits characters placed in the output queue. Line disciplines generally use the *t_rawq* and *t_canq* (canonicalized queue) for input; in line mode, the canonical queue contains full lines, and the raw queue contains any current partial line. Line disciplines that do not use *t_rawq* and *t_canq* sometimes overlay this area with a different form of receive buffer.
- Hardware and software modes and parameters. The *t_flags* word contains modes set by both TIOCSETP and TIOCLSET. The line speeds set by TIOCSETP are placed in *t_ispeed* and *t_ospeed*.
- The line-discipline number (*t_line*).
- Special characters. The characters that require special actions are all stored in *t_tchars*. These characters are set with TIOCSETP (erase and kill), TIOCSETC (interrupt, quit, start, stop, end-of-file, and alternate end-of-line), and TIOCSLTC (suspend, delayed suspend, reprint-line, flush-output, word-erase, and escape-next-character).
- Hardware driver information. This information includes *t_oproc*, the driver procedure that starts transmissions after data are placed in the output queue; *t_addr*, the address of hardware-related data structures or registers; and *t_dev*, the device number of the terminal line.

- Terminal line-discipline software state. This includes the terminal column number and counts for tab and erase processing (*t_col*, *t_rocount* and *t_rocol*), the process group of the terminal (*t_pgrp*), and the identity of any processes selecting for input or output (*t_rsel* and *t_wsel*).
- Terminal or window size (*t_winsize*). This information is not used by the kernel, but it is stored here to present a consistent interface. In addition, 4.3BSD supplies the SIGWINCH signal (derived from Sun Microsystems' SunOS) that can be sent when the size of a window changes. This is useful for windowing packages such as X Window System [Scheifler & Gettys, 1986] that allow users to resize windows dynamically; programs such as text editors running in such a window need to be informed that something has changed and that they should recheck the window size.

The *tty* structure is initialized by the hardware terminal driver's open routine and by the line-discipline open routine.

9.5 Process Groups and Terminal Control

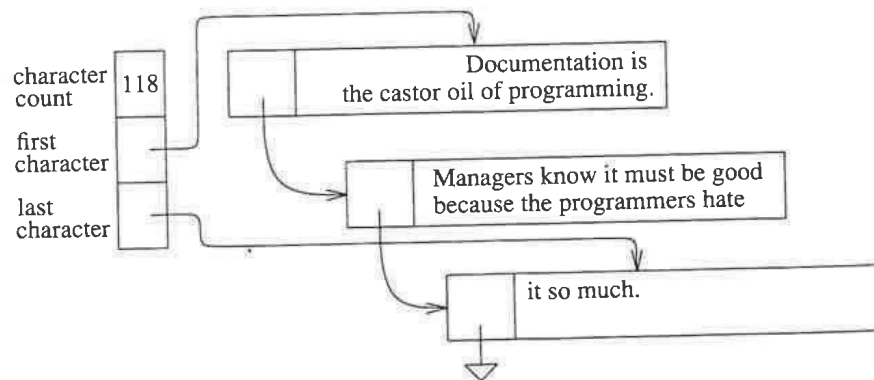
The process-control (job-control) facilities described in Chapter 4 depend on the terminal I/O system to control access to the terminal. Each job (a group of processes that are controlled together) is known by a process-group number. Each terminal structure contains a process-group number as well. The shell and the terminal are assigned to a process group (the shell's PID) when the terminal is first opened. Signals that are generated by characters typed at the terminal are sent to all the processes in the terminal's current process group. Some of those signals cause the process group to stop. The shell sets the process-group number for each job to be the PID of one of the group's members; each time it places a new job in the foreground, the shell sets the terminal process group to that of the new process group. Thus, the terminal process group is the identifier for the process group that is currently in control of the terminal—that is, for the process group running in the *foreground*. Other process groups may run in the *background*. If a process in such a group attempts to read from the terminal, its process group is sent another signal, which stops the process group. Optionally, background processes that attempt terminal output may be stopped as well. Each process holds a pointer to its controlling terminal (if it has one); these rules for control of input and output operations apply to only those operations on the controlling terminal.

Earlier systems without job control used the process group in a different way; similar behavior occurs in 4.3BSD when a login shell that does not support job control is used. The shell and the terminal are initially assigned to the same process group. New processes started by such a shell remain in the same group. A process group therefore identifies all the processes initiated from the same login shell when job control is not in use.

9.6 C-lists

The terminal I/O system deals with data in blocks of widely varying sizes. Most input and output operations deal with single characters (typed input characters and their output echoes). Input characters are usually aggregated with previous input to form lines of varying sizes. Some output operations involve larger numbers of data, such as screen updates or other command output. The data structures originally designed for terminal drivers, the character block, *C-block*, and character list, *C-list*, are still in use in 4.3BSD. Each C-block is a fixed-size, aligned buffer that contains a linkage pointer and space for buffered characters. Its size is always a power of two, so that the system can compute boundaries between blocks by masking off the low-order bits of a pointer. 4.3BSD uses 64-byte C-blocks storing 60 characters; earlier systems used buffers of 32 bytes. A queue of input or output characters is described by a C-list, which contains pointers to the first and last characters, and a count of the number of characters in the queue (see Fig. 9.1). Both of the pointers point to characters stored in C-blocks. When a character is removed from a C-list queue, the count is decremented and the pointer to the first character is incremented. If the pointer has advanced beyond the end of the first C-block on the queue, the pointer to the next C-block is obtained from the forward pointer at the start of the current C-block. After the forward pointer is updated, the empty C-block is placed on a free chain. A similar process is used to add a character to a queue. If there is no room in the current buffer, another buffer is allocated from the free list, the linkage pointer of the last buffer is set to point at the new buffer, and the tail pointer is set to the first storage location of the new buffer. The character is stored where indicated by the tail pointer, the tail pointer is incremented, and the character count is incremented. A set of utility routines manipulates C-lists: *getc()* removes the next character from a C-list and returns it; *putc()* adds a character to the end of a C-list. Groups of characters may be added to or removed from C-lists with *b_to_q()* and *q_to_b()*, respectively. The terminal driver also requires the abilities to remove a character

Figure 9.1 A C-list structure.



from the end of a queue with *unputc()*, to examine characters in the queue with *nextc()*, and to concatenate queues with *catq()*.

When UNIX was developed on computers with small address spaces, the design of buffers for the use of terminal drivers was a challenge. The C-list and C-block provided an elegant solution to the problem of storing arbitrary-length queues of data for terminal input and output queues when the latter were designed for machines with small memories. On machines such as the VAX that have larger address spaces, it would be better to use a data structure that uses less CPU time per character at a cost of reduced space efficiency. The VAX still uses the original C-list data structure because of the high labor cost of converting to a new data structure; a change to the queue structure would require modification of all the line disciplines and all the terminal device drivers, which would be a substantial amount of work. One could just change the implementations of the interface routines, but the routines would still be called once per character unless the actual interface was changed, and changing the interface would require changing the drivers.

9.7 RS-232 and Modem Control

Most terminals and modems are connected via asynchronous RS-232 serial ports. This type of connection supports several lines in addition to those that transmit and receive data. UNIX usually supports only a few of these lines. The most commonly used lines are those showing that the equipment on each end is ready for data transfer. The RS-232 electrical specification is asymmetrical; each line is driven by one of the two devices connected and is sampled by the other device. Thus, one end in any normal connection must be wired as data terminal equipment (DTE), such as a terminal, and the other as data communications equipment (DCE), such as a modem. Note that *terminal* in DTE means *endpoint*: A terminal on which people type is a DTE, and a computer also is a DTE. The data terminal ready (DTR) line is the output of the DTE end that serves as a *ready* indicator. In the other direction, the data carrier detect (DCD) line indicates that the DCE device is ready for data transfer. VAX terminal interfaces are all wired as DTE (they may be directly connected to modems, or connected to local terminals with null modem cables). Therefore, the terminology used in the 4.3BSD terminal drivers and commands reflects this orientation, even though many computers incorrectly use the opposite convention.

When terminal devices are opened, the DTR output is asserted so that the connected modem or other equipment may begin operation. If *modem control* is supported on a line, the open does not complete, and no data are transferred until the DCD input *carrier* is detected. Thus, an open on a line connected to a modem will block until a connection is made, which commonly occurs when a call is received from a remote modem. Data may then be transferred as long as carrier remains on. If the modem loses the connection, the DCD line is turned off, and subsequent reads and writes fail.

Ports that are used with local terminals or other DTE equipment are connected with a *null modem* cable that connects DTR on each end to DCD on the other end.

Alternatively, the DTR output on the host port can be *looped back* to the DCD input. Finally, the driver may be configured to ignore modem-control inputs.

9.8 Terminal Operations

Now that we have examined the overall structure of the terminal I/O system and have described that system's data structures, as well as the hardware it controls, we continue with a description of its operation. We shall examine the operation of a generalized terminal hardware device driver and the normal terminal line discipline. We shall not cover the autoconfiguration routines present in each driver, as they function in the same way as do those described in Chapter 8.

Open

Each time the special file for a terminal-character device is opened, the hardware driver's open routine is called. The open routine checks that the requested device was configured into the system and was located during autoconfiguration, then initializes the *tty* structure. If the device was not yet open, the default modes and line speed are set. The *tty* state is set to *TS_WOPEN*, waiting for open. Then, if the device supports modem-control lines, the open routine enables the DTR output line, and blocks awaiting assertion of the DCD input line. Most drivers support *device flags* to override modem control; these flags are set in the system configuration file, and are stored in the driver data structures. If the bit corresponding to a terminal line number is set in a device's flags, modem control lines are ignored on input. When a carrier signal is detected on the line, the *TS_CARR_ON* bit is set in the terminal state. The driver then passes control to the initial (or current) line discipline through its open entry.

The default line discipline when a device is first opened is discipline 0, the *old* terminal driver. Its open routine, *tyopen()*, checks to see whether the process opening the line has been assigned to a process group. If not, the process is assigned to the process group of the terminal, *t_pgrp*. If the terminal has no process group either, it uses the PID of the opening process. In this way, the initial open of a terminal line by the program accepting new logins sets the process groups of the process and of the terminal to the same value. If the line was not already open, the terminal-size information for the line is set to zero, indicating an unknown size. The line is then marked as open (state bit *TS_OPEN*).

Output Line Discipline

After a line has been opened, a write on the resulting file descriptor produces output to be transmitted on the terminal line. Writes to character devices result in calls to the device write entry, *d_write*, with a device number and a *uio* structure describing the data to be written. Terminal hardware drivers use the device number to locate the correct *tty* structure, then call the line discipline *l_write* entry with the *tty* structure and *uio* structure as parameters.

The line-discipline write routine does most of the work of output translation and flow control. It is responsible for copying data into the kernel from the user

process calling the routine and for placing the translated data onto the terminal's output queue for the hardware driver. The terminal-driver write routine, *ttwrite()*, first checks that the terminal line still has carrier asserted; if not, an error is returned. Then, it checks whether the current process should be allowed to write to the terminal immediately. The user may set a *tty* option to allow only the foreground process to do output. If this option is in use, and if the terminal line is the controlling terminal for the process, then the process may do output immediately only if it is in the foreground process group (i.e., if the process groups of the process and of the terminal are the same). If the process is prevented from doing output, a signal (SIGTTOU) is sent to the process group of the process to suspend it. When the user moves the process group to the foreground, the write will be attempted again.

When it has confirmed that the write will be allowed, *ttwrite()* enters a loop that copies the data to be written into the kernel, checks for any output translation that is required, and places the data on the output queue for the terminal. It prevents the queue from becoming overfull by blocking if the queue fills before all characters have been processed. The limit on the queue size (the *high watermark*) is dependent on the output line speed. When it must wait for output to drain before proceeding, *ttwrite()* sets a flag in the *tty* structure state, TS_ASLEEP, so that the transmit-complete interrupt handler will awaken it when the queue is reduced to the *low watermark* (also dependent on line speed). The check of the queue size and subsequent sleep must be done such that the interrupt is guaranteed to occur after the sleep; see Fig. 9.2.

Once errors, permissions, and flow control have been checked, *ttwrite()* copies the user's data into a local buffer in chunks of at most 100 characters using *uiomove()*. If it is doing character mapping for single-case terminals, each character is translated and placed on the output queue with *ttyoutput()*; otherwise, the buffer is processed in sections. In raw mode, no translations are done, and the entire buffer is processed at once. The *new* terminal driver also provides an option (LITOUT) to allow a full eight-bit character output path. In other modes, groups of characters requiring no translation are located using *scanc()* with a lookup table that marks motion characters that might need translation or delay. Each group is placed into the output queue using *b_to_q()*. Trailing special

Figure 9.2 Checking the output queue in a line discipline.

```

struct tty *tp;

s = spltty();
if (tp->t_outq.c_cc > high-water-mark) {
    ttstart(tp);
    tp->t_state |= TS_ASLEEP;
    sleep(&tp->t_outq, TTOPRI);
}
splx(s);

```


characters are output with *ttyoutput()*. In either case, *ttwrite()* must check that sufficient C-list blocks are available; if not, it waits for a short time (by sleeping on *lbolt* for up to 1 second), then retries.

The routine that does output with translation is *ttyoutput()*, which accepts a single character, processes it as necessary, and places the result on the output queue. The following translations may be done, depending on the terminal mode:

- Tabs are expanded to spaces for terminals that do not support them.
- On upper-case-only terminals, backslashes are prepended to capital letters, and lower-case characters are replaced with capitals.
- Newlines are replaced with carriage return plus line feed if the CRMOD option is set. (This option also controls mapping of input carriage returns to newlines.)
- Delays are appended to characters that produce printhead motion. Delays are stored as characters outside of the ASCII character range, with the most significant bit set and the remaining bits specifying the amount of delay.

After data are placed on the output queue of a *tty*, *ttstart()* is called to initiate output. Unless output is already in progress, or has been suspended by receipt of a *stop* character or an output delay, *ttstart()* calls the hardware-driver start routine specified in the *tty*'s *t_oproc* field. If all the data have been processed and have been placed into the output queue, *ttwrite()* then returns an indication that the write completed successfully, and the actual serial character transmission is managed asynchronously by the device driver.

Output Top Half

The device driver handles the hardware-specific operation of character transmission, as well as synchronization and flow control for output. The structure of the *start()* routine varies little from one driver to another. There are two general classes of output mechanisms, depending on the type of hardware device. The first class operates on devices that are capable of direct-memory-access (DMA) output, which can fetch the data directly from the C-list block. (The DEC DH-11 terminal multiplexer is an example of this type of interface.) DMA interfaces supported by 4.3BSD on the VAX are all connected to the UNIBUS; thus, the output buffer must be mapped to UNIBUS address space with map registers in the UNIBUS adapter (see Chapter 8). To avoid the overhead of map allocation on each transmission, UNIX places the C-list blocks in a contiguous area of physical memory, then maps this area to the UNIBUS when the first terminal device is opened. The physical-memory base address of the transfer is computed from the UNIBUS address of the mapped area plus the offset of the data into the C-list area.

The driver start routine uses *ndqb()* to compute the number of contiguous characters in the first C-list block of the output queue that can be transmitted in one operation. If the terminal is in a mode that allows output delays, *ndqb()* is given a mask to distinguish delay characters. If a delay character is the first character to be transmitted, the terminal line is marked with *TS_TIMEOUT*, and *timeout()* is used

to schedule the end of the delay, calling *ttstrt()* to resume output when the delay ends. Otherwise, the UNIBUS address and the number of characters in the data block are computed, and the device is programmed with this information. The device fetches the data from main memory, transmits each of the characters in turn, and interrupts the CPU when the transmission is complete.

In the second class of terminal interfaces are those that transmit a single character at a time; the DEC DZ-11 terminal multiplexer is such a device. Each character is loaded into the device's output-character register for transmission. The CPU must then wait for the transmit-complete interrupt before sending the next character. Because of the large number of interrupts generated in this mode of operation, much of the information needed at interrupt time is computed in advance. The information needed is a pointer to the next character to be transmitted, to the extent of the block of characters to be transmitted, and to the address of the hardware device register to receive the next character. This strategy is known as *pseudo-DMA*; the precomputed information is stored in a *pdma* structure. A small assembly-language routine receives each hardware transmit-complete interrupt, transmits the next character, and returns. When there are no characters left to transmit, it calls a specified C-language interrupt routine with an indication of the line that completed transmission. The normal driver thus sees the illusion of DMA output, as it is not called until the entire block of characters has been transmitted.

After an output operation is initiated, the terminal state is marked with *TS_BUSY* so that new transmissions will not be attempted until the current one completes.

Output Bottom Half

When transmission of a block of characters has been completed, the hardware multiplexer interrupts the CPU; the transmit interrupt routine is then called with the unit number of the device. In most cases, the device has a register that the driver can read to determine which of the device's lines have completed transmit operations. For each line that has finished output, the interrupt routine clears the *TS_BUSY* flag. The characters that have been transmitted are removed from the output queue with *ndflush()*. These steps complete one section of output.

The line-discipline start routine is called to initiate the next operation; as noted, this routine generally does nothing but call the driver start routine specified in the terminal *t_oproc* field. The start routine now checks to see whether the output queue has been reduced to the low watermark, and, if it has been, whether the top half is waiting for space in the output queue. If the *TS_ASLEEP* flag is set, the output process is awakened; if a process is recorded in *t_wsel* as selecting for output, it is notified with *selwakeup()*. Then, if the output queue is not empty, the next operation is started as before.

Input Bottom Half

Unlike output, terminal input is not initiated by a system call, but arrives asynchronously when the terminal line receives characters from the keyboard or other input

device. Thus, the input processing in the terminal system occurs mostly at interrupt time. Most hardware multiplexers interrupt each time a character is received on any line. They usually provide a *silo* that stores received characters, along with the line number on which the characters were received and any associated status information, until the device handler retrieves the characters. Use of the silo prevents characters from being lost if the CPU has not processed a received-character interrupt by the time the next character arrives. On many devices, the system can avoid per-character interrupts by programming the device to interrupt only after the silo is partially or completely full. However, the driver must then check the device periodically so that characters do not stagnate in the silo if additional input does not trigger an interrupt. Input flow-control characters must be processed without much delay, and users may notice delay in echoing of characters as well. If the device can also be programmed to interrupt a short time after the first character enters the silo, the driver may avoid periodic checks of the silo as well. The drivers in 4.3BSD for devices with such timers always use the silo interrupts. Other terminal drivers use per-character interrupts until the input rate is high enough to warrant the use of the silo alarm and a periodic scan of the silo.

When a device receiver interrupt occurs, or when a timer routine detects input, the receiver-interrupt routine reads each character from the input silo along with the latter's line number and status information. Normal characters are passed as input to the terminal line discipline for the receiving *tty* through its *l_rint* entry:

```
(*linesw[tp->t_line].l_rint)(input-character, tp);
```

Characters received with hardware-detected parity errors are discarded if only one parity is expected (either even or odd). When a *break* condition is detected (a longer-than-normal character with only 0 bits), an interrupt character or a null is passed to the line discipline, depending on the terminal mode.

The receiver-interrupt (*l_rint*) routine for the normal terminal line discipline is *ttyinput()*. The interpretation of terminal input described in Section 9.1 is done here. Input characters are echoed if desired. In raw mode, characters are placed into the raw input queue without interpretation. Otherwise, most of the work done by *ttyinput()* is to check for characters with special meanings and to take the indicated actions. Other characters are placed into the raw queue. In cooked mode, if the received character is a carriage return or another character that causes the current line to be made available to the program reading the terminal, the contents of the raw queue are added to the canonicalized queue, and *ttwakeup()* is called to notify any process waiting for input. In other modes, *ttwakeup()* is called when each character is processed. It will awaken any process sleeping on the raw queue awaiting input for a *read*, and will notify processes selecting for input. If the terminal has been set for signal-driven I/O using *fcntl* and the *FASYNC* flag, a *SIGIO* signal is sent to the process group controlling the terminal.

Ttyinput() must also check that the input queue does not become too large, exhausting the supply of C-list blocks; input characters are dropped when the limit (255 characters) is reached. If the *TANDEM* option is set, end-to-end flow control is invoked when the queue reaches half full by output of a stop character (normally *XOFF* or control-S).

Up to this point, all processing is asynchronous, and occurs whether or not a *read* call is pending on the terminal device. In this way, type-ahead is allowed to the limit of the input queues.

Input Top Half

Eventually, a *read* call is made on the file descriptor for the terminal device. Like all calls to read from a character-special device, this one results in a call to the device driver's *d_read* entry with a device number and a *uio* structure describing the destination for the data. Terminal device drivers use the device number to locate the *tty* structure for the device, then call the line discipline *l_read* entry to process the system call.

The *l_read* entry for the terminal driver is *ttread()*. Like *ttwrite()*, *ttread()* first checks that the terminal line still has carrier, returning an error if not. It then checks to see whether the current process is in the process group that currently controls the terminal. If the process group of the current process is not, that process group is stopped with the signal SIGTTIN until its process group receives control of the terminal. Finally, *ttread()* checks for data in the appropriate queue (the canonicalized queue in cooked mode, the raw queue in raw or cbreak mode). If no data are present, *ttread()* returns the error EWOULDBLOCK if the terminal is using nonblocking I/O; otherwise, it sleeps on the address of the raw queue. When *ttread()* is awakened, it restarts processing from the beginning, as the terminal state or process group might have changed while it was asleep.

When characters are present in the queue for which *ttread()* is waiting, they are removed from the queue one at a time with *getc()* and are copied out to the user's buffer with *wreadc()*. In raw mode, the characters are not examined as they are processed. In other modes, certain characters receive special processing as they are removed from the queue. Carriage returns are translated to newlines if the CRMOD flag is set, and the delayed-suspension character causes the current process group to be stopped with signal SIGTSTP. In addition, in cooked or cbreak modes, the end-of-file character terminates the read without being passed back to the user program. If there was no previous character, this condition results in the read returning zero characters, which is interpreted by user programs as indicating end-of-file. Characters are processed and returned to the user until the character count in the *uio* structure reaches zero, the queue is exhausted, or (in cooked mode) a line terminator is reached. When the call returns, the returned character count will be the amount by which the requested count was decremented as characters were processed.

After the read completes, if the terminal is in TANDEM mode and a stop character was sent because the queue was filling up, a start character (normally XON, control-Q) is sent if the queue is now less than 20 percent full.

The stop Routine

Character output on terminal devices is done in blocks as large as possible for efficiency. However, there are two events that should cause a pending output operation to be stopped. The first event is the receipt of a stop character, which should stop output as quickly as possible; in some cases, the device receiving

output is a printer or other output device with a limited buffer size. The other event that stops output is the receipt of a special character that causes output to be discarded, possibly due to a signal. In either case, the terminal line discipline calls the character device driver's *d_stop* entry to stop any current output operation. Two parameters are provided, a *tty* structure and a flag that indicates whether output is to be flushed or suspended. If flushing output, the terminal discipline removes all the data in the output queue after calling the device stop routine.

The implementation of the *d_stop* routine is hardware-dependent. Different drivers stop output by disabling the transmitter, thus suspending output, or by changing the current character count to zero. Pseudo-DMA drivers may change the limit on the current block of characters so that the pseudo-DMA routine will call the transmit-complete interrupt routine after the current character is transmitted. Most drivers set a flag in the *tty* state, *TS_FLUSH*, when a stop is to flush data and the aborted output operation will cause an interrupt. When the transmit-complete interrupt routine runs, it notes the *TS_FLUSH* flag, and avoids updating the output-queue character count (the queue has probably already been flushed by the time the interrupt occurs). If output is to be stopped but not flushed, the *TS_TTSTOP* flag is set in the *tty* state; the driver must stop output such that the latter may be resumed from the current position.

The *ioctl* Routine

Section 9.3 described the user interface to terminal drivers and line disciplines, most of which is accessed via the *ioctl* system call. Most of these calls manipulate software options in the terminal line discipline; some of them also affect the operation of the asynchronous serial port hardware. In particular, the hardware line speed, word size, and parity are derived from these settings. Accordingly, *ioctl* calls are processed both by the current line discipline and by the hardware driver.

The device driver *d_ioctl* routine is called with a device number, an *ioctl* command, and a pointer to a data buffer when an *ioctl* is done on a character-special file. Like the read and write routines, most terminal-driver *ioctl* routines locate the *tty* structure for the device, then pass control to the line discipline. Depending on the return value from the line discipline, the driver may do additional processing, as shown in Fig. 9.3. The line-discipline *ioctl* routine does most of the common commands, including change of line discipline and change of terminal parameters. The *ioctl* routine returns an error number if an error is detected, or returns zero if the command has been processed successfully; it returns a minus one if the command is not recognized. In the first two cases, the hardware driver checks to see whether the command might affect the programming of the serial port parameters. If it might, the driver must reset the hardware parameters according to the new mode. The driver then returns the error indication from the line discipline. Some drivers implement additional *ioctl* commands that manipulate modem-control outputs. These commands are not recognized by the line discipline, and thus must be handled by the driver.


```

error = (*linesw[tp->t_line].l_ioctl)(tp, cmd, data, flag);
if (error >= 0) {
    if (cmd == TIOCSETP || cmd == TIOCSETN ||
        cmd == TIOCLBIS || cmd == TIOCLBIC ||
        cmd == TIOCLSET)
        dhpam(unit); /* reset hardware parameters */
    return (error);
}
switch (cmd) {
    /* do hardware-specific commands */
}

```

Figure 9.3 Handling an error return from a line discipline.

Modem Transitions

The manner in which UNIX uses modem-control lines on terminal lines was introduced in Section 9.7. Most terminal multiplexers support at least the set of modem-control lines used by 4.3BSD; those that do not act instead as if carrier were always asserted. When a device is opened, the DTR output is enabled, and then the state of the carrier input is checked. If the state of the carrier input changes later, this change must be detected and processed by the driver. Some devices have a separate interrupt that reports changes in modem-control status; others report such changes along with other status information with received characters. One device used on the VAX (the DZ-11) does not interrupt when modem-control lines change, and the driver must check their status periodically. When a change is detected, the line discipline is notified by a call to its `l_modem` routine with the new state of the carrier input.

The normal terminal-driver modem routine, `ttymodem()`, maintains the state of the `TS_CARR_ON` flag in the `tty` structure, and processes corresponding state changes. When carrier establishment is detected, a wakeup is issued for any process waiting for an open to complete. When carrier drops on an open line, the process group controlling the terminal is sent a hangup signal, `SIGHUP`, as well as a continue signal (`SIGCONT`) for any processes that are stopped, and the terminal queues are flushed. The return value of `ttymodem()` indicates whether the driver should maintain its DTR output. If the value is zero, DTR should be turned off. `Ttymodem()` also implements an obscure terminal option to use the carrier line for flow-control handshaking, stopping output when carrier drops and resuming when it returns.

Closing Terminal Devices

When the last reference to a terminal device is closed, the device-driver close routine is called. Both the line discipline and the hardware driver may need to close down gracefully. The line-discipline close routine is called first. The standard line-discipline close entry, `tylclose()`, waits for any pending output to drain if

carrier is still on, then flushes the input queue and sets the terminal line discipline to the default. Note that the close may be interrupted by a signal while waiting for output to complete. The hardware driver may clear any pending operations, such as transmission of a break. If the state bit `TS_HUPCLS` has been set with the `TIOCHPCL` *ioctl*, DTR is disabled to hang up the line. Finally, *ttyclose()* flushes all the queues and clears all the terminal state.

9.9 Other Line Disciplines

We have examined the operation of the terminal I/O system using the standard terminal-oriented line-discipline routines, which implement both the *new* Berkeley line discipline and the *old* Seventh Edition UNIX line discipline. We now describe several other line disciplines in the system for completeness. Note that the preceding discussion of the operation of the terminal multiplexer drivers applies when these disciplines are used, as well as when the terminal-oriented disciplines are used.

Berknet

The first example of an alternate line discipline for alternate uses of serial lines is the Berknet line discipline. *Berknet* is an obsolete batch-oriented network that was used to connect PDP-11 and VAX UNIX systems using 9600-baud serial lines. Due to the overhead of input processing in the standard line discipline, a special reduced-function network discipline was devised. A seven-bit data path is provided for transfer of blocks of data terminated by newlines; acknowledgment and checksumming occur outside of the system. The Berknet discipline allocates a single buffer for each line, which it uses to buffer a single line of input at a time (the network software acknowledged each line before another could be sent). The *l_rint* received-character routine is expanded in-line in most of the multiplexer drivers to reduce the overhead as much as possible.

Serial Line IP Discipline

Another line discipline, *Serial Line IP* or SLIP, is used by more modern networking software to encapsulate and transfer Internet Protocol (IP) datagrams over asynchronous serial lines [Romkey, 1988]. (See Chapters 11 and 12 for information about IP.) The *slattach* program opens a serial line, sets the line's speed, and enters the SLIP line discipline. The SLIP line-discipline open routine associates the terminal line with a preconfigured network interface and prepares to send and receive network packets. Once the interface's network address is set with the *ifconfig* program, the network will route packets through the SLIP line to reach the system to which it connects. Packets are framed with a simple scheme; a framing character (0300 octal) is used to separate packets. Framing characters that occur within packets are quoted with an escape character (0333 octal) and are trans-

posed (to 0334 octal). Escape characters within the packet are escaped and transposed (to 0335 octal).

The output path is started every time a packet is output to the SLIP interface. The SLIP discipline places the framing character and the data of the packet onto the output queue of the *tty*, escaping framing and escape characters as needed. It then starts transmission by calling *ttstart()*, which in turn calls the device's start routine referenced in the *tty t_oproc* field. It may place multiple packets onto the output queue before returning, as long as there is sufficient room in the queue and the system is not running short of C-list blocks. When transmission completes, the device driver calls the SLIP start routine, which continues to place data onto the output queue until all packets have been sent or the queue fills up again.

When characters are received on a line that is using the SLIP discipline, escaped characters are translated and data characters are placed into a network buffer. When a framing character ends the packet, the packet is presented to the network, and the buffer is reinitialized.

The SLIP discipline allows moderate-speed network connections to machines without specialized high-speed network hardware.

Graphics Tablet Discipline

The *tablet line discipline* is used to connect graphic devices such as digitizing tablets to the system using a serial line. Once the discipline is entered, it receives graphics data from the device continuously, and allows the application program to poll for the most recent information by reading from the line. The format of the information returned is dependent on that provided by the device; several different formats are supported.

9.10 Summary

We have examined the terminal I/O system in 4.3BSD. Terminal handling is split between the device drivers for the character devices to which terminals are connected and the line disciplines that are shared among all such drivers. The standard terminal line discipline supports both the *new* Berkeley terminal discipline and the *old* Seventh Edition UNIX discipline. Other line disciplines are used to support nonterminal uses of serial lines, such as network connections or connections to digitizing hardware. Planned future work, as mentioned in Chapter 1, includes support of a POSIX line discipline.

Exercises

- 9.1 What are the three possible modes of terminal input? Which mode is most commonly in use when users converse with an interactive screen editor?

- 9.2 Why are there two character queues for dealing with terminal input? Describe the use of each.
- 9.3 What do we mean when we say that modem control is supported on a terminal line? How are lines of this sort typically used?
- 9.4 What signals are sent to the processes associated with a terminal if a user disconnects the modem line in the middle of a session?
- 9.5 How is the high watermark on a terminal's output queue determined?
- 9.6 Why does a DH-11 terminal multiplexer have significantly less overhead than does a DZ-11?
- *9.7 Consider a facility that allowed a tutor on one terminal to monitor and assist students working on other terminals. Everything the students typed would be transmitted both to the system as input, and to the tutor's terminal as output. Everything the tutor typed would be directed to the students' terminals as input. Describe how this facility might be implemented with a special-purpose line discipline. Describe further useful generalizations of this facility.
- *9.8 The terminal line discipline supports logical erasure of input text when characters, words, and lines are erased. Remembering that other system activities continue while a user types an input line, explain what complications must be considered in the implementation of this feature. Name three exceptional cases and describe their effects on the implementation.
- **9.9 What are the advantages of the use of line disciplines by device drivers for terminal multiplexers? What are the limitations? Propose an alternative approach to the current structure of the terminal I/O system.
- **9.10 Consider the differences between the Berknet and SLIP disciplines. Although both use alternate line disciplines for efficiency, one of these uses would be extremely difficult to implement without modifications to the kernel, such as a specialized line-discipline module. Which use would be difficult to implement? Explain your choice.

References

Romkey, 1988.

J. Romkey, "A Nonstandard for Transmission of IP Datagrams Over Serial Lines: SLIP," RFC 1055, SRI Network Information Center, Menlo Park, CA (June 1988).

Scheifler & Gettys, 1986.

R. W. Scheifler & J. Gettys, "The X Window System," *ACM Transactions on Graphics* 5(2), pp. 79-109 (April 1986).

PART 4

Interprocess Communication

Interprocess Communication

Historically, UNIX has been weak in the area of *interprocess communication*. Prior to the release of 4.2BSD, the only standard interprocess-communication facility found in UNIX was the *pipe*, a reliable, flow-controlled, byte stream that could be established only between two related processes on the same machine. The limiting nature of pipes inspired many experimental facilities such as the Rand Corporation UNIX system's *ports* [Sunshine, 1977], *multiplexed files* that were an experimental part of Version 7 UNIX [UPMV7, 1983], and the Accent IPC facility developed at Carnegie-Mellon University [Rashid, 1980]. Some communication facilities were developed for use in application specific versions of UNIX; for example, the shared memory, semaphores, and message queues that were part of the *Columbus UNIX System*. The requirements of the DARPA research community, which drove much of the design and development of 4.2BSD, resulted in a significant effort to address the lack of a comprehensive set of interprocess-communication facilities in UNIX. The facilities designed and implemented in 4.2BSD were refined following that version's release. As a result, 4.3BSD provides a rich set of interprocess-communication facilities intended to support the construction of *distributed programs* built on top of communications primitives.

The interprocess-communication facilities are described in this chapter. The layer of software that implements these facilities is strongly intertwined with the network subsystem. The architecture of the network system is described in Chapter 11, and the networking **protocols** themselves are examined in Chapter 12. The reader will find it easiest to **understand the material** in these three chapters by first reading Chapter 10, and then Chapters 11 and 12. A section devoted to tying everything together appears at the end of Chapter 12.

10.1 Interprocess-Communication Model

There were several goals in the design of the interprocess-communication enhancements to UNIX. The most immediate need was to provide access to communication networks such as the DARPA Internet [Cerf, 1978]. Previous work in providing network access had focused on the implementation of the network protocols, exporting the transport facilities to UNIX applications via special-purpose, and often awkward, character-device interfaces [Cohen, 1977; Gurwitz, 1981]. As a result, each new network implementation resulted in a different application interface requiring most existing programs to be significantly altered or completely rewritten. The 4.2BSD interprocess-communication facilities were intended to provide a sufficiently general interface to allow network-based applications to be constructed independently of the underlying communication facilities.

The second goal was to allow multiprocess programs, such as distributed databases, to be implemented. The UNIX *pipe* requires all communicating processes to be derived from a common parent process. The use of pipes forced systems such as the Ingres database system to be designed with a somewhat contorted structure [Kalash *et al.*, 1986]. New communication facilities were needed to support communication between unrelated processes residing locally on a single host computer and residing remotely on multiple host machines.

Finally, the emerging networking and workstation technology required that the new communication facilities allow construction of local-area network services such as file servers. The intent was to provide facilities that could be easily used in supporting resource sharing in a distributed environment; the intention was not to construct a distributed UNIX system.

The interprocess-communication facilities were designed to support the following:

- **Transparency:** Communication between processes should not depend on whether or not the processes are on the same machine.
- **Efficiency:** The applicability of any interprocess-communication facility is limited by its performance. In 4.2BSD, interprocess communication is layered on top of network communication for performance reasons. The alternative is to provide network communication as a service accessed via the interprocess-communication facilities. Although this design is more modular, it would have required that network-communication facilities be accessed through one or more server processes. At the time 4.2BSD was designed, the prevalent hardware on which the system ran had such a slow process context-switch time that the performance of the communication facilities in a distributed environment would have been seriously constrained. Thus, the most efficient implementation of interprocess-communication facilities layers interprocess communication on top of network-communication facilities.
- **Compatibility:** Existing naive UNIX processes should be usable in a distributed environment without change. A *naive process* is characterized as a process that

performs its work by reading from the standard input file and writing to the standard output file. A *sophisticated process* is one that manages other processes or uses knowledge about specific devices, such as a terminal. A major reason why UNIX has been successful is its support for modularity through the use of naive processes that act as byte-stream filters. Although sophisticated applications such as shells and screen editors exist, they are far outnumbered by the collection of naive application programs.

In the course of designing the interprocess-communication facilities, the developers identified the following requirements to support these goals, and they developed a unifying concept for each:

- The system must support communication networks that use different sets of protocols, different naming conventions, different hardware, and so on.
The notion of a *communication domain* was defined for these reasons. A communication domain embodies the standard semantics of communication and naming. Different networks almost always have different standards for specifying the name of a communication endpoint. Names may also vary in their properties. In one network, a name may be a fixed address for a communication endpoint, whereas in another it may be used to locate a process that can move between locations. The semantics of communication can include the cost associated with the reliable transport of data, the support for multicast transmissions, the ability to pass access rights or capabilities, and so on. By distinguishing communication properties, applications can select a domain appropriate to their needs.
- A unified abstraction for an endpoint of communication is needed that can be manipulated with a UNIX file descriptor.
The *socket* is the abstract object from which messages are sent and received. Sockets are created within a communication domain much as files are created within a filesystem. Unlike files, however, sockets exist only as long as they are referenced.
- The semantic aspects of communication must be made available to applications in a controlled and uniform manner. That is, applications must be able to request styles of communication, such as virtual circuits or datagrams, but these styles must be provided in a manner that is consistent across all communication domains.
All sockets are *typed* according to their communication semantics. Types are defined by the subset of properties a socket supports. These properties are
 1. In-order delivery of data
 2. Unduplicated delivery of data
 3. Reliable delivery of data
 4. Preservation of message boundaries

5. Support for out-of-band messages
6. Connection-oriented communication

Pipes have the first three properties, but not the fourth. An out-of-band message is one that is delivered to the receiver outside the normal stream of incoming, in-band data, and it usually is associated with an urgent or exceptional condition. A connection is a mechanism used to avoid having to transmit the identity of the sending socket with each packet of data. Instead, the identity of each endpoint of communication is exchanged prior to transmission of any data, and is maintained at each end so that it can be presented at any time. A *datagram socket* models potentially unreliable, connectionless packet communication; a *stream socket* models a reliable connection-based byte stream that may support out-of-band data transmission; and a *sequenced packet socket* models sequenced, reliable, unduplicated connection-based communication that preserves message boundaries. Other types of sockets are desirable and can be added.

- Processes must be able to locate endpoints of communication so that they may rendezvous without being related; for this reason, sockets can be *named*. A socket's name is meaningfully interpreted only within the context of the communication domain in which the socket is created. The names used by most applications are human-readable strings. However, the name for a socket that is used within a communication domain is usually a low-level *address*. Rather than placing name-to-address translation functions in the kernel, 4.3BSD provides functions for application programs to use in translating names to addresses. In the remainder of this chapter, we refer to the name of a socket as an *address*.

Using Sockets

Using sockets is reasonably straightforward. First, a socket must be created with the *socket* system call:

```
s = socket(domain, type, protocol);  
int s, domain, type, protocol;
```

The type of socket is selected according to the characteristic properties required by the application. For example, if reliable communication is required, a stream socket might be selected. The *type* parameter is one of the socket types defined in a system header file. The *domain* parameter specifies the communication domain (or *protocol family*, see Section 10.4) in which the socket should be created; this domain is dependent on the environment in which the application is working. The most common domain for intermachine communication is the Internet communication domain, because of the large number of hosts that support the Internet communication protocols. The last parameter, the *protocol*, can be used to indicate a specific communication protocol to use in supporting the socket's operation.

Protocols are indicated by well-known constants specific to each communication domain. If the protocol is specified as zero, the system picks an appropriate protocol. The *socket* system call returns a file descriptor (a small integer number, see Section 6.2) that is then used in later socket operations. The *socket* call is similar to *open*, except that it creates a new instance of an object of the specified type, whereas *open* creates a new reference to an existing object, such as a file or device.

After a socket has been created, the next step depends on the type of socket being used. The most commonly used type of socket requires a *connection* before it can be used. Creation of a connection between two sockets usually requires that each socket have an address bound to it. Applications may explicitly specify a socket's address or may permit the system to assign one. A socket's address is immutable. Socket addresses may be reused if the communication domain permits, although domains normally ensure that a socket address is unique on each host, so that the association between two sockets is unique within the communication domain. The address to be bound to a socket must be formulated in a *socket address structure*. Applications find addresses of well-known services by looking up their names in a database. The format of addresses can vary among domains; to permit a wide variety of different formats, the system treats addresses as variable-length byte arrays, which are prefixed with a tag that identifies their format. The call to bind an address to a socket is

```
error = bind(s, addr, addrlen);
int error, s;
struct sockaddr *addr;
int addrlen;
```

where *s* is the descriptor returned from a previous *socket* system call.

For several reasons, binding a name to a socket was separated from creating a socket. First, sockets are potentially useful without names. If all sockets had to be named, users would be forced to devise meaningless names without reason. Second, in some communication domains, it may be necessary to supply additional, nonstandard information to the system before binding a name to a socket—for example, the “type of service” required when using a socket. If a socket's name had to be specified at the time the socket was created, supplying this information would not be possible without further complicating the interface.

In connection-based communication, the process that initiates a connection normally is termed a *client process*, whereas the process that receives, or responds to, a connection is termed a *server process*. In the *client process*, a connection is initiated with a *connect* system call:

```
error = connect(s, serveraddr, serveraddrlen);
int error, s;
struct sockaddr *serveraddr;
int serveraddrlen;
```

In the *server process*, the socket is first marked to indicate incoming connections are to be accepted on it:

```
error = listen(s, backlog);  
int error, s, backlog;
```

Connections are then received, one at a time, with

```
snew = accept(s, clientaddr, clientaddrlen);  
int snew, s;  
struct sockaddr *clientaddr;  
int *clientaddrlen;
```

The *backlog* parameter in the *listen* call is used to specify an upper bound on the number of pending connections that should be queued for acceptance. Processes can obtain the address of the client along with the new connected socket in the *accept* call by specifying the *clientaddr* and *clientaddrlen* parameters. Note that *accept* returns a file descriptor associated with a *new* socket. This new socket is the socket through which client-server communication can take place. The original socket *s* is used solely for managing the queue of connection requests in the server.

Sockets that are not connection-based may also use the *connect* system call to fix a peer's address, although this is not required. The system calls available for sending and receiving data (described later) permit connectionless sockets to be used without a fixed peer address via specification of the destination with each transmitted message. Likewise, connectionless sockets need not bind an address to a socket before using the socket to transmit data. However, in some communication domains, addresses are assigned to sockets when the latter are first used, if no specific address was bound.

A variety of calls are available for sending and receiving data. The usual *read* (*readv*) and *write* (*writv*) system calls, as well as the newer *send* and *recv* system calls, can be used with sockets that are in a connected state. *Send* and *recv* differ from the more common interface in that they both support an additional *flags* parameter. The *flags* can be used to *peek* at incoming data on reception (*MSG_PEEK*), to send or receive out-of-band data (*MSG_OOB*), and to send data without network routing (*MSG_DONTROUTE*). The *sendto* and *recvfrom* system calls have all the capabilities of *send* and *recv* and in addition permit callers to specify or receive the address of the peer with whom they are communicating; these calls are most useful for connectionless sockets, where the peer may vary on each message transmitted or received. Finally, the *sendmsg* and *recvmsg* system calls support the full interface to the interprocess-communication facilities. Besides scatter-gather operations being possible, an address may be specified or received, the optional flags described previously are available, and specially interpreted data, called access rights, may be transmitted and received (see Fig. 10.1).

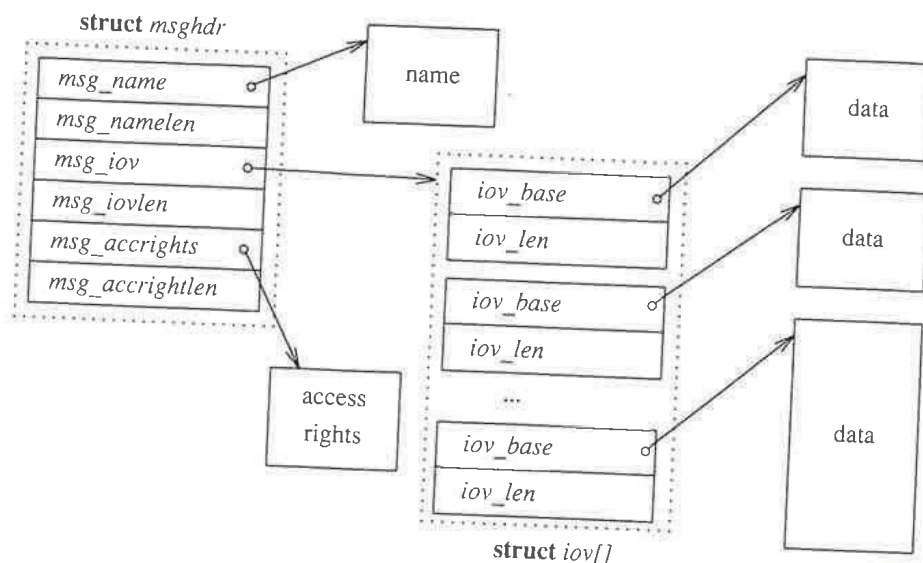


Figure 10.1 Data structures for the `sendmsg` and `recvmsg` system calls.

In addition to these system calls, several other calls are provided to access miscellaneous services. The `socketpair` call provides a mechanism by which two connected sockets may be created without binding addresses. This facility is almost identical to a pipe except for the potential for bidirectional flow of data. In fact, pipes are implemented internally as a pair of sockets. The `getsockname` call returns the locally bound address of a socket, whereas the `getpeername` call returns the address of the socket at the remote end of a connection. The `shutdown` call is used to terminate data transmission or reception at a socket, and two `ioctl`-style calls—`setsockopt` and `getsockopt`—can be used to set and retrieve various parameters that control the operation of a socket or of the underlying network protocols. These options include the ability to transmit broadcast messages, to set the size of a socket's send and receive data buffers, and to await the transmission of queued data when a socket is destroyed. Sockets are discarded with the normal `close` system call.

The interface to the interprocess-communication facilities was purposely designed to be orthogonal to the standard UNIX system interfaces—that is, to the `open`, `read`, and `write` system calls. This decision was made to avoid overloading the familiar interface with undue complexity. In addition, the developers thought that using an interface that was completely independent of the UNIX filesystem would improve the portability of software because, for example, UNIX pathnames would not be involved. Backward compatibility, for the sake of naive processes, was still deemed important; thus, the familiar read-write interface was augmented to permit access to the new communication facilities wherever it made sense (e.g., when connected stream sockets were used).

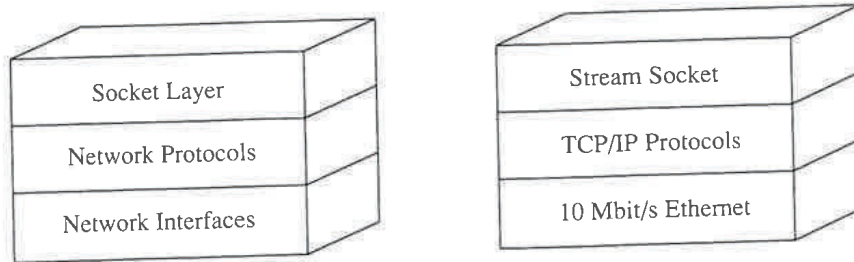


Figure 10.2 Interprocess-communication implementation layering.

10.2 Implementation Structure and Overview

The interprocess-communication facilities are layered on top of the networking facilities, as shown in Fig. 10.2. Data flows from the application through the socket layer to the networking support, and vice versa. State required by the socket level is fully encapsulated in the socket layer, whereas any protocol-related state is maintained in auxiliary data structures that are specific to the supporting protocols. Responsibility for storage associated with transmitted data is passed from the socket level to the network level. Consistent adherence to this rule assists in simplifying details of storage management. Within the socket layer, the socket data structure is the focus of all activity. The system-call interface routines manage the actions related to a UNIX system call, collecting the system-call parameters (see Section 3.2) and converting user data into the format expected by the second-level routines. Most of the socket abstraction is implemented within

Table 10.1 Socket-layer support routines.

Routine	Function
<i>screate()</i>	create a new socket
<i>sobind()</i>	bind a name to a socket
<i>solisten()</i>	mark a socket as listening for connection requests
<i>soaccept()</i>	accept a pending connection on a socket
<i>soconnect()</i>	initiate a connection to another socket
<i>soconnect2()</i>	create a connection between two sockets
<i>sosend()</i>	send data
<i>soreceive()</i>	receive data
<i>soshutdown()</i>	shut down data transmission or reception
<i>sosetopt()</i>	set the value of a socket option
<i>sogetopt()</i>	get the value of a socket option

the second-level routines. All second-level routines have names with a *so* prefix, and directly manipulate socket data structures and manage the synchronization between asynchronous activities; these routines are listed in Table 10.1.

The remainder of this chapter focuses on the implementation of the socket layer. Section 10.3 discusses how memory is managed at the socket level and below in the networking subsystem; Section 10.4 covers the socket and related data structures; Section 10.5 presents the algorithms for connection setup; Section 10.6 discusses data transfer; and Section 10.7 describes connection shutdown. Throughout this chapter, references to the supporting facilities provided by the network-communication protocols are made with little elaboration; a complete description of the interaction between the network protocols and the socket layer appears in Chapter 11, and the internals of the network protocols are presented in Chapter 12.

10.3 Memory Management

The requirements placed on a memory-management scheme by interprocess-communication and network protocols tend to be very different from those of other parts of the operating system. Although all require the efficient allocation and reclamation of memory, communication protocols in particular need memory in widely varying sizes. Memory is needed for fixed-sized data structures, such as sockets and protocol state blocks, as well as for variable structures such as communication protocol packets. A special-purpose memory-management facility was created for use by the interprocess-communication and networking systems to address these needs.

Mbufs

The memory-management facilities revolve around a data structure called an *mbuf* (see Fig. 10.3). Mbufs, or memory buffers, are 128 bytes long, with 112 bytes of this space reserved for data storage. For large messages, up to 1 Kbyte of data may be associated with an mbuf by referencing a page from a private page pool. Data are stored either in the internal data area or in the external page, never in both. Data associated with an mbuf are accessed by the addition of an offset field in the mbuf structure to the base address of the mbuf. The offset is between 12 and 124 for data contained within the mbuf; for data located in a mapped page, the (unsigned) offset is greater than 124.

In addition to the offset field used to reference the data associated with an mbuf, a length field is also maintained. The length field indicates the number of bytes of valid data to be found at the offset. The offset and length fields allow routines to trim data efficiently at the start or end of an mbuf. In deletion of data at the start of an mbuf, the offset is incremented and the length is decremented. In deletion of data at the end of an mbuf, only the length need be decremented. When space is available within an mbuf, data can be added at either end. This

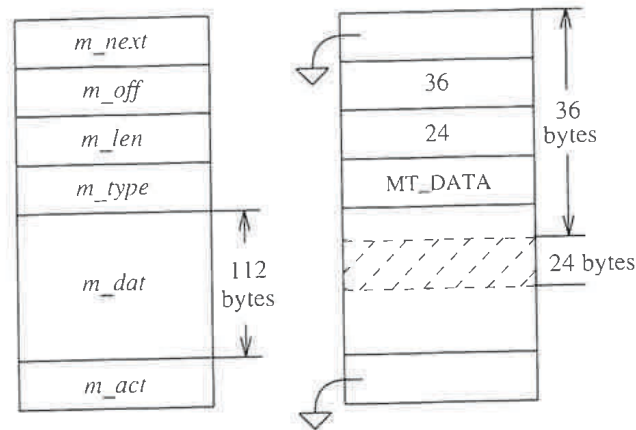


Figure 10.3 Memory-buffer (mbuf) data structure.

flexibility to add and delete space without copying is particularly useful in implementing communication protocols. Protocols routinely strip protocol information off the front or back of a message before the message's contents are handed to a higher-level processing module, or add protocol information as a message is passed to lower levels.

The provision for mapping pages into an mbuf permits data to be moved without a memory-to-memory copy operation. Instead, the page-table entries associated with the physical pages of memory can be changed to remap the memory from one mbuf to another. In addition, when multiple copies of a block of data are required, the same page can be mapped into multiple mbufs to avoid physical copies; an array of reference counts is maintained for the pool of mapped pages to support this style of **sharing**.

Multiple mbufs can be **linked** to hold an arbitrary number of data. This linkage is accomplished with the *m_next* field of the mbuf. By convention, a chain of mbufs linked in this manner is treated as a single object. For example, the communication protocols construct packets from chains of mbufs. A second field, *m_act*, is used to link objects constructed from chains of mbufs into lists of objects. Throughout our discussions, a collection of mbufs linked together with the *m_next* field will be called a *chain*; chains of mbufs linked together with the *m_act* field will be called a *list*.

The final component of the mbuf structure is the type field. Each mbuf is typed according to its use, with unused mbufs marked MT_FREE. The mbuf type serves two purposes. The only operational use of the type is to distinguish optional components of a message in **an mbuf chain that is queued for reception** on a socket data queue. Otherwise, the type information is used in maintaining statistics about storage use and, in case of problems, as an aid in tracking mbufs.

Mbufs have fixed-sized, rather than variable-sized, data areas for several reasons. First, the fixed size minimizes **memory fragmentation**. This was an important consideration at the time the networking software was originally designed, as

one of the targeted machines was the BBN C70, a machine with a 20-bit physical address space. Second, communication protocols are frequently required to prepend or append headers to existing data areas, to split data areas, or to trim data from the beginning or end of a data area. The mbuf facilities are designed to handle such changes without reallocation or copying whenever possible. Finally, the *dtom()* function described later would be much more expensive if mbufs were not fixed in size.

The ease of use and efficiency with which the mbuf memory-management scheme operates has resulted in the use of mbufs for data structures other than those for which they were intended. Mbufs are used to hold everything from variable-sized protocol packets to a zombie process's status. Certain system routines even go so far as to impose a substructure on an mbuf's data-storage area to minimize wasted space. This overloading of the mbuf facility is due more to the weakness of the system's memory-allocation facilities than to the strength of the mbuf storage mechanism. Many vendors have addressed the lack of a general heap storage facility within the system, freeing them to eliminate many of the inappropriate uses of mbufs. Similar changes are also present in post-4.3BSD systems.

Storage-Management Algorithms

The mbuf store is composed of three central resources: a pool of pages allocated from the system memory allocator, a private page map for mapping pages into kernel address space, and an array of counters used in maintaining reference counts on mapped pages. Pages allocated to the mbuf store are divided into mbuf structures or are used for storage of data too numerous to fit in the mbuf structure's data array. Two free lists are maintained, one for mbuf structures and one for mapped pages. When the system is booted, the mbuf-allocation routines initialize the store by allocating 4 Kbyte of physical memory to create 32 mbuf structures, and four pages to initialize the pool of mappable pages. Further memory may be allocated as the system operates, up to a maximum of 256 Kbyte. Each page allocated to the mbuf store is recorded in the private page map *Mbmap*. Mbuf-allocation requests indicate that they must be fulfilled immediately or that they can wait for available resources. If a request is marked as "can wait" and the requested resources are unavailable, the process is put to sleep to await available resources. The nonblocking allocation request is necessary for code that executes at interrupt level. In the event that the mbuf store is exhausted and system resources are unavailable, or that the store has already been expanded to its limit, the mbuf-allocation routines ask the network-protocol modules to give back any available resources they can. Memory allocated to the mbuf store is never returned to the system.

An mbuf-allocation request is made through a call to *m_get()*, *m_getclr()*, or one of the equivalent macros used for efficiency purposes. If an mbuf is available on the free list, it is simply taken; otherwise, an attempt is made to expand the resource pool by one page. If the system has resources available and the mbuf store is not already at its limit, then the newly allocated page is divided into

Table 10.2 Socket types supported by the system.

Name	Type	Properties
SOCK_STREAM	stream	reliable, sequenced, data transfer; may support out-of-band data
SOCK_DGRAM	datagram	unreliable, unsequenced, data transfer with message boundaries preserved
SOCK_SEQPACKET	sequenced packet	reliable, sequenced, data transfer with message boundaries preserved
SOCK_RAW	raw	direct access to the underlying communication protocols

domain. If the communication domain is unable to support the type of socket requested (i.e., no suitable protocol exists), the request will fail.

Sockets are described by a *socket* data structure that is dynamically created at the time of a *socket* system call. Communication domains are described by a *domain* data structure that is statically defined within the system based on the system's configuration (see Section 13.5). Communication protocols within a domain are described by a *protosw* structure that is also statically defined within the system for each protocol implementation configured. When a request is made to create a socket, the system uses the name of the communication domain to search linearly the list of configured domains. If the domain is found, the domain's table of supported protocols is consulted for a protocol appropriate for the type of socket being created, or for a specific protocol requested. (A wildcard entry may exist for a raw domain.) Should multiple protocol entries satisfy the request, the first is selected. We shall begin discussion of the data structures by examining the *domain* structure. The *protosw* structure is discussed in Section 11.1.

Communication Domains

The *domain* structure is shown in Fig. 10.4. The *dom_name* field is the ASCII name of the communication domain.¹ The *dom_family* field identifies the *protocol family* used by the domain; possible values are shown in Table 10.3. Protocol families refer to the suite of communication protocols of a domain used to support the communication semantics of a socket. The *dom_protosw* field points to the table of protocols supported by the communication domain, whereas the remaining entries contain pointers to domain-specific routines used in the management and transfer of access rights (described later).

¹ In the original design, communication domains were to be specified with ASCII strings; they are now specified with manifest constants.

<i>dom_family</i>	PF_UNIX
<i>dom_name</i>	"unix"
<i>dom_init</i>	...
<i>dom_externalize</i>	<i>unp_externalize()</i>
<i>dom_dispose</i>	<i>unp_dispose()</i>
<i>dom_protosw</i>	<i>unixsw</i>
<i>dom_protoswNPROTOSW</i>	<i>&unixsw[5]</i>
<i>dom_next</i>	

Figure 10.4 Communication-domain data structure.

Sockets

The *socket* data structure is shown in Fig. 10.5. A *socket* data structure resides in the data area of an mbuf. Whereas the use of an mbuf allows *socket* structures to be dynamically created, the size of the mbuf imposes a limit on the size of the

Table 10.3 Protocol families.

Name	Description
PF_UNIX	local communication; e.g., pipes
PF_INET	DARPA Internet (TCP/IP)
PF_IMPLINK	1822 Input Message Processor link layer
PF_PUP	old Xerox network
PF_CHAOS	MIT Chaos network
PF_NS	Xerox Network System (XNS) architecture
PF_NBS	National Bureau of Standards (NBS) network
PF_ECMA	European Computer Manufacturers network
PF_DATAKIT	AT&T Datakit network
PF_CCITT	CCITT protocols, e.g., X.25
PF_SNA	IBM System Network Architecture (SNA)
PF_DECnet	DEC network
PF_DLI	direct link interface
PF_LAT	local-area network terminal interface
PF_HYLINK	Network Systems Corporation Hyperchannel (raw)
PF_APPLETALK	AppleTalk network

mbufs, which are placed on the free list. The caller then allocates the needed mbuf from the free list.

Releasing mbuf resources is straightforward; *m_free()* frees a single mbuf, and *m_freem()* frees a chain of mbufs. The system easily detects pages associated with mbufs when an mbuf is released by checking the offset field in the mbuf structure when the mbuf is freed. Pages are placed onto the free page list when their reference count reaches zero.

Mbuf Utility Routines

Many useful utility routines exist for manipulating mbufs. Those routines that will be used in Chapter 11 are described briefly here.

The *m_copy()* routine makes a copy of an mbuf chain starting at a logical offset, in bytes, from the start of the data. This routine may be used to copy all or only part of a chain of mbufs. If an mbuf is composed of mapped pages, the copy will reference the same pages by incrementing the reference counts on the pages; otherwise, the data portion is copied as well.

The *m_adj()* routine adjusts the data in an mbuf chain by a specified number of bytes, shaving data off either the front or back. No data are ever copied; *m_adj()* operates purely by manipulating the offset and length fields in the mbuf structures.

The *mtod()* routine takes a pointer to an mbuf header and a data type and returns a pointer to the data in the mbuf, cast to the given type. The *dtom()* function is the inverse: It takes a pointer to an arbitrary address in the data of an mbuf and returns a pointer to the mbuf header (not to the head of the mbuf chain). In 4.3BSD, this operation is done through simple truncation of the data address to an mbuf-sized boundary.

The *m_pullup()* routine is used to rearrange an mbuf chain so that a specified number of data resides in a contiguous data area within the mbuf (not page). This operation is used so that objects such as protocol headers are contiguous and can be treated as normal data structures, and so that *dtom()* will work when the object is freed. If there is room, *m_pullup()* will also add up to MPULL_EXTRA extra bytes to the contiguous region in an attempt to avoid being called in the future.

10.4 Data Structures

Sockets are the basic object used by communicating processes. A socket's type defines the communication semantics, whereas the communication domain defines auxiliary properties important to the use of the socket. Table 10.2 shows the four types of sockets currently supported by the system. To create a new socket, applications must specify the socket type and communication domain in which the socket is to be created. The request may also indicate a specific network protocol to be used by the socket. If no protocol is indicated, the system selects an appropriate protocol from the set of protocols supported by the communication

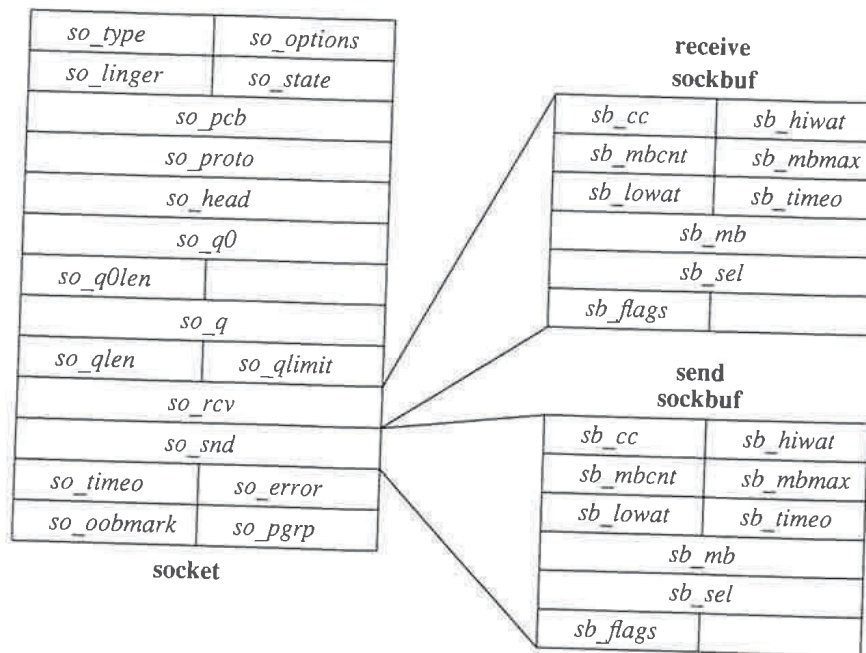


Figure 10.5 Socket data structure.

structure. Sockets contain information about their type, the supporting protocol in use, and their state (Table 10.4). Data being transmitted or received are queued at the socket in the form of a list of mbuf chains. Various fields are present for managing queues of sockets created during connection establishment. Each socket structure also holds a process-group identifier. The process-group identifier is used in delivering the SIGURG and SIGIO signals; SIGURG is sent when an urgent condition exists for a socket, and SIGIO is used by the asynchronous I/O facility (see Section 6.2). The socket contains an error field analogous to

Table 10.4 Socket states.

State	Description
SS_NOFDREF	no file-table reference
SS_ISCONNECTED	socket connected to a peer
SS_ISCONNECTING	in process of connecting to peer
SS_ISDISCONNECTING	in process of disconnecting from peer
SS_CANTSENDMORE	cannot send more data to peer
SS_CANTRCVMORE	cannot receive more data from peer

the error number in the user structure (Section 4.2); this field is needed for reporting asynchronous errors to the owner of the socket.

Sockets are located through a process's file descriptor via the file table. When a socket is created, the *f_data* field of the file structure is set to point at the socket structure, and the *f_ops* field to point to the set of routines defining socket-specific file operations. In this sense, the socket structure is a direct parallel of the *inode* structure used by the filesystem.

The socket structure acts as a queueing point for data being transmitted and received. As data enter the system as a result of system calls, such as *write* or *send*, the socket layer passes the data to the networking subsystem as a chain of mbufs for immediate transmission. If the supporting protocol module decides to postpone transmission of the data, or if a copy of the data is to be maintained until an acknowledgment is received, the data are queued in the socket's transmit buffer. When the network has consumed the data, it discards them from the outgoing queue. On reception, the network passes data up to the socket layer, also in mbuf chains, where they are then queued until the application makes a system call to request them. To avoid resource exhaustion, sockets impose upper bounds on the number of data that can be queued in a socket data buffer. This *high watermark* is initially set by the protocol, although an application can change the value up to a maximum of 56 Kbyte. The network protocols can examine the high watermark and use the value in flow-control policies, but they never alter it once it is set. A *low watermark* is also present in each socket data buffer, although it is not currently used. The low watermark is intended to allow applications to control data flow by specifying a minimum number of data to be present in the data buffer before a reception request is satisfied. For output, it would set the minimum amount of space available before transmission can be attempted.

Sockets used to accept incoming connection requests maintain two other queues. The list of sockets headed by the *so_q0* field represents a queue of connections that are incomplete at the communication protocol level, whereas the *so_q* field heads a list of sockets for which a completed connection exists at the protocol level. Like the data queue, the queue of completed connections also has an application-controllable limit on it. It is important to recognize that, although a connection may be established by the network protocol, the application may choose not to accept the established connection, or it may immediately close down the connection after discovering the identity of the client. Unlike some other systems, this system has no facility to allow an application to control the acceptance of a connection at the network level. We could consider this to be a deficiency, although in practice its principal effect is to require a higher-level handshake between the client and server processes before the server process will provide service. It is also possible for a network protocol to delay completion of a connection until after the application has obtained control with the *accept* system call.

Socket Addresses

Sockets may be labeled so that processes can connect to them. The socket layer treats an address as an opaque object. Applications supply and receive addresses

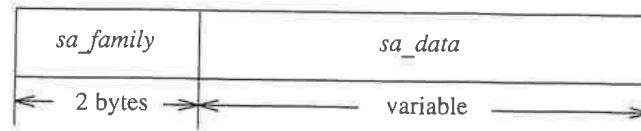
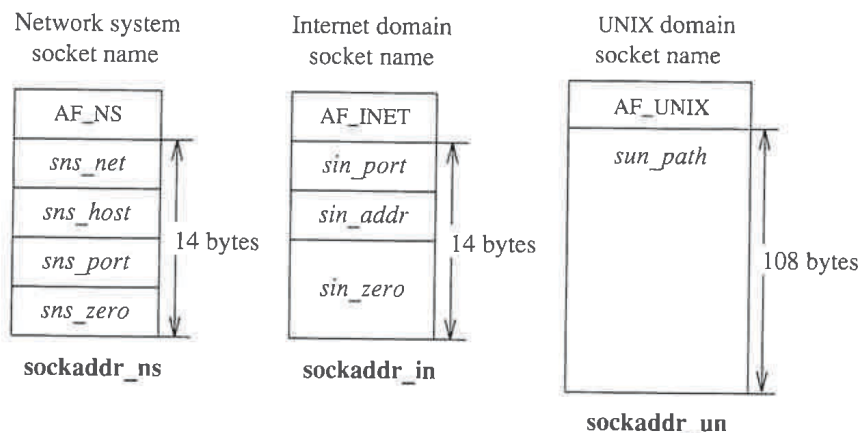


Figure 10.6 Socket address template structure.

as tagged, variable-length byte strings. Addresses always reside in an mbuf on entry to the socket layer. A structure called a *sockaddr*, shown in Fig. 10.6, may be used as a template for referring to the identifying tag of each address. Each address family implementation includes routines for address family-specific operations on addresses. When addresses must be manipulated—for example, to compare them for equality—a pointer to the address (a *sockaddr* structure) is used to extract the address family tag. This tag is then used to identify the routine to invoke for the desired operation.

It is common for addresses passed in by an application to reside in mbufs only long enough for the socket layer to pass them to the supporting protocol for transfer into a fixed-sized address structure. This occurs, for example, when a protocol records an address in a protocol state block. The *sockaddr* structure is the common means by which the socket layer and network-support facilities exchange addresses. The size of the generic data array was chosen to be large enough to hold most addresses directly. Communication domains that support larger addresses may ignore the array size. Only the routing data structures (discussed in Section 11.5) contain fixed-sized generic *sockaddr* structures. The UNIX communication domain, for example, stores filesystem pathnames in mbufs and allows socket names as large as 108 bytes, as shown in Fig. 10.7. The

Figure 10.7 Network system, Internet, and UNIX address structures.



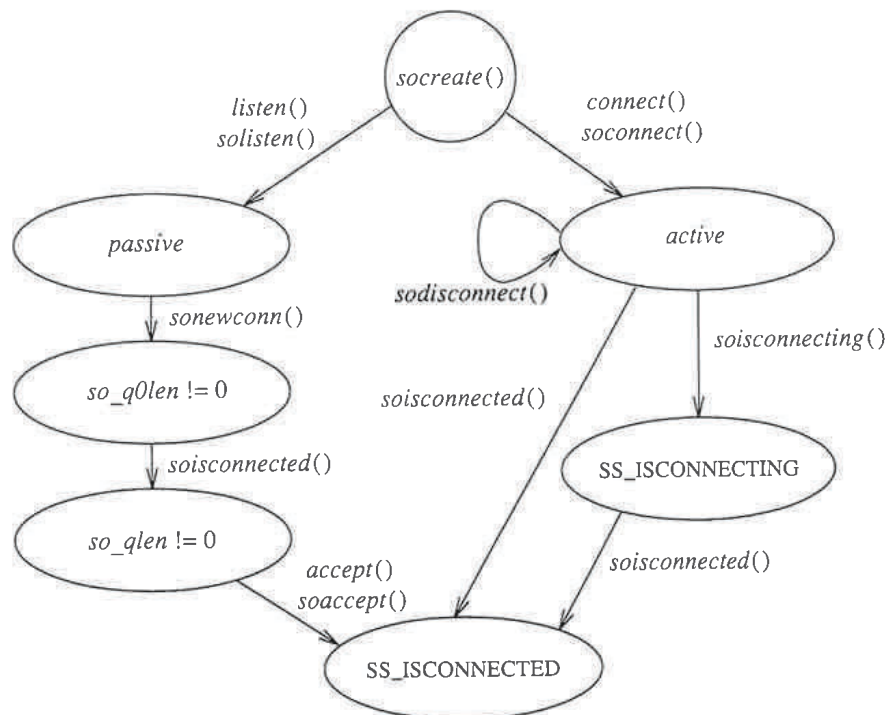
Internet communication domain, on the other hand, uses a structure that combines a DARPA Internet address and a port number. The Internet protocols reserve space for addresses in an Internet control-block data structure, and free up mbufs that contain addresses after copying their contents.

10.5 Connection Setup

For two processes to pass information between them, an *association* must be established. The steps involved in creating an association (*socket*, *connect*, *listen*, *accept*, etc.) were described in Section 10.1. In this section, we shall study the operation of the socket layer in establishing associations. As the state associated with a connectionless transfer of data is fully encapsulated in each message that is sent, our discussion shall focus on connection-based associations established with the *connect*, *listen*, and *accept* system calls.

Connection establishment in the client-server model is asymmetric. A client process actively initiates a connection to obtain service, whereas a server process passively accepts connections to provide service. Fig. 10.8 shows the state-transition diagram for a socket used to initiate or accept connections. State

Figure 10.8 Socket state transitions during process rendezvous.

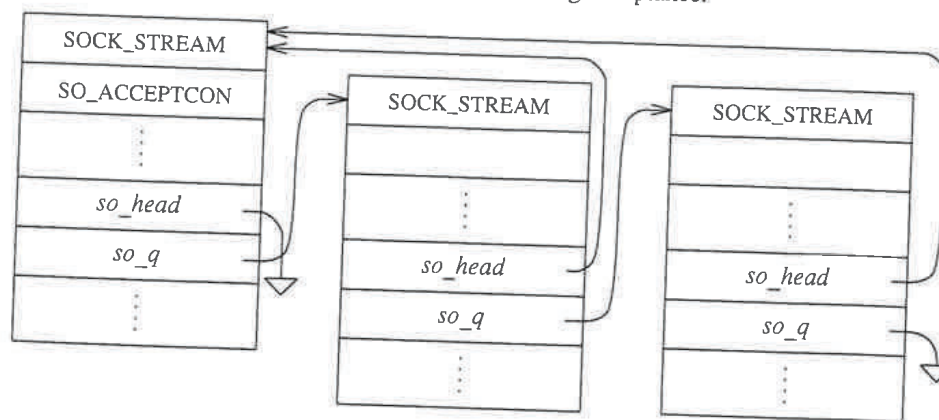


transitions are initiated either by user actions (i.e., system calls) or by protocol actions that result from receiving network messages or servicing timers that expire.

Sockets are normally used to send and receive data. When they are used in establishing a connection, they are treated somewhat differently. If a socket is to be used to accept a connection, a *listen* system call must be used. The *listen* call invokes *solisten()*, which notifies the supporting protocol that the socket will be receiving connections, establishes an empty list of pending connections at the socket (through the *so_q* field), and then marks the socket as *accepting connections*, *SO_ACCEPTCON*. At the time a *listen* is done, a backlog parameter is specified by the application. This parameter is used to set a limit on the number of incoming connections the system will queue awaiting acceptance by the application. (The system enforces a maximum on this limit.) Once a socket is set up to receive connections, the remainder of the work in creating connections is managed by the protocol layers. For each connection established at the server side, a new socket is created with the *sonewconn()* routine. These new sockets may be placed on the socket's queue of partially established connections while the connections are being completed, but once ready they are moved to the queue of connections completed and pending acceptance by an application (see Fig. 10.9). When an *accept* system call is made to obtain a connection, the system verifies that a connection is present on the socket's queue of incoming connections. If no pending connection is present, the system puts the process to sleep until one arrives (unless nonblocking I/O is being used with the socket, in which case an error is returned). When a connection is available, the associated socket is removed from the queue, a new file descriptor is allocated to reference the socket, and the result is returned to the caller. If the *accept* call indicates that the peer's identity is to be returned, the peer's address is copied into the supplied buffer.

On the client side, an application requests a connection with the *connect* system call, supplying the address of the peer socket to which to connect. The

Figure 10.9 Connections queued at a socket awaiting acceptance.



system verifies that a connection attempt is not already in progress for that socket, then invokes *soconnect()* to initiate the connection. The *soconnect()* routine first checks the socket to see whether the latter is already connected. If the socket is already connected, the existing connection is first terminated (this is done with only datagram sockets). With the socket in an unconnected state, *soconnect()* then marks the state as connecting, and makes a request to the protocol layer to initiate the new connection. Once the connection request has been passed to the protocol layer, if the connection request is incomplete, the system puts the process to sleep to await notification by the protocol layer that a completed connection exists. A nonblocking connect may return at this point, but a process awaiting a completed connection will awaken only when the connection request has been completed—either successfully or with an error condition.

A socket's state during connection establishment is managed jointly by the socket layer and the supporting protocol layer. The socket's state value is never altered directly by a protocol; to promote modularity, all modifications are performed by surrogate socket-layer routines, such as *soisconnected()*. Errors that are detected asynchronously are communicated to a socket in its *so_error* field. For example, if a connection request fails because the protocol layer detects that the requested service is unavailable, the *so_error* field usually is set to *ECONNREFUSED* before the requesting process is awakened. The socket layer always inspects the value of *so_error* on return from a call to *sleep()*; this is the standard method used in identifying errors detected asynchronously by the protocol layers.

10.6 Data Transfer

Most of the work performed by the socket layer is in sending and receiving data. It should be noted that the socket layer itself explicitly refrains from imposing any structure on data transmitted or received via sockets. This is in contrast to some interprocess-communication facilities available for UNIX today [Fitzgerald & Rashid, 1986]. Within the overall interprocess-communication model, any data interpretation or structuring is logically isolated in the implementation of the communication domain. An example of this logical isolation is the ability to pass file descriptors between processes using UNIX domain sockets.

Sending and receiving of data can be done with any one of several system calls. The system calls vary according to the amount of information to be transmitted and received and the state of the socket being used to perform the operation. For example, the *write* system call may be used with a socket that is in a connected state, as the destination of the data is implicitly specified by the connection; but the *sendto* or *sendmsg* system calls allow the process to specify the destination for a message explicitly. Likewise, when receiving data, the *read* system call allows a process to receive data on a connected socket without receiving the sender's address; the *recvfrom* and *recvmsg* system calls allow the process to retrieve the incoming message and the sender's address. The decision to provide many different system calls rather than to provide only a single general interface

is debatable. It would have been possible to implement a single system-call interface and to provide simplified interfaces to applications via user-level library routines. Internally, all transmission and reception requests are converted to a uniform format and are passed to the socket-layer *sendit()* and *recvit()* routines, respectively.

Transmitting Data

The *sendit()* routine is responsible for gathering all system-call parameters that the application has specified into the kernel's address space (except the actual data), and then invoking the *sosend()* routine to perform the transmission. *Sosend()* handles most of the socket-level data-transmission options. This includes requests for transmission of out-of-band data and of data without network routing. In addition, *sosend()* is responsible for putting processes to sleep when their data transmissions exceed the buffering available in the socket's send buffer. The actual transmission of data is performed by the supporting communication protocol; *sosend()* moves data from the user's address space into mbufs in the kernel's address space, and then makes calls to the protocol to transfer the data.

Most of the work performed by *sosend()* is involved with breaking up an application's transmission request into one or more protocol transmission requests. This is necessary only when the number of data in the user's request plus the number of data queued in the socket's send data buffer exceeds the socket's high watermark. It is not permissible to break up a request if message boundaries are to be maintained in the user's data stream, as each request made by the socket layer to the protocol modules implicitly indicates a boundary in the data stream. Honoring each socket's high watermark ensures that a protocol will always have space in the socket's send buffer to queue unacknowledged data. It also ensures that no process, or group of processes, can monopolize system resources.

For sockets that guarantee reliable data delivery, a protocol will normally maintain a copy of all transmitted data in the socket's send queue until receipt is acknowledged by the receiver. Protocols that provide no assurance of delivery normally accept data from *sosend()* and directly transmit the data to the destination without keeping a copy. But *sosend()* itself does not distinguish between reliable and unreliable delivery.

Sosend() always ensures that a socket's send buffer has sufficient space available to store the next section of data to be transmitted. If a socket has insufficient space in its send buffer to hold all the data to be transmitted, *sosend()* uses the following strategy: If more than one page of data is present in the send queue, more than one page of data is awaiting transmission, and the process is not using nonblocking I/O, put the process to sleep until more space is available in the send buffer; otherwise, formulate a protocol transmit request according to the available space in the send buffer. This strategy tends to preserve the application-specified message size and helps to avoid fragmentation at the network level. The latter is very important, as system performance is significantly

better when data-transmission units are a multiple of the machine's page size, because of the optimized support for mbufs with mapped pages. (*Sosend()* will copy data from the user's address space into pages allocated from the mbuf page pool whenever the data will fill at least one-half of a page.)

The *sosend()* routine, in manipulating a socket's send data buffer, takes care to ensure that access to the buffer is synchronized among multiple sending processes. It does this by bracketing accesses to the data structure with calls to *sblock()* and *sbunlock()*. Interlocking against asynchronous network activity is also a concern here, as the network-protocol modules that operate at network-interrupt level cannot wait for access to a data structure such as a socket data buffer. Thus, they do not honor the locking protocol used between processes. To block network-protocol modules, *sosend()* must raise the processor priority level to *splnet* to ensure that no protocol processing takes place that might alter the state of a socket being manipulated.

Receiving Data

The *soreceive()* routine is used to receive data queued at a socket. As the counterpart to *sosend()*, *soreceive()* appears at the same level in the internal software structure and performs similar tasks. Two types of data normally are queued for reception at a socket: in-band data and access rights. Out-of-band data are not normally placed in a socket's receive buffer, but instead are managed in the protocol layer and are retrieved through a special interface when requested by the user. (This special handling allows varying styles of urgent data transmission.)

Soreceive() checks the socket's state, including the receive data buffer, for incoming data, errors, or state transitions, and then processes queued data according to their type and the actions specified by the caller. A system-call request may indicate that only out-of-band data should be retrieved (MSG_OOB), or that data should be returned but not removed from the data buffer (by specifying the MSG_PEEK flag).

Data present in the receive data buffer are organized in one of two ways, depending on whether or not message boundaries are preserved. If the socket preserves message boundaries on incoming data, the buffer is organized as a list of messages. Each mbuf chain on a list represents a single message, and the boundary is implicit in the list structure (see Fig. 10.10). Protocols that supply the sender's address with each message place a single mbuf containing the address at the front of message. Immediately following any address is an optional mbuf containing access rights. Regular data mbufs follow the access rights. Names and access rights are distinguished by the type field in an mbuf; addresses are marked as **MT_SONAME**, whereas rights are **tagged** as **MT_RIGHTS**. When message boundaries are not preserved, the receive data buffer is simply a single chain of mbufs. As before, mbuf types are used to distinguish optional pieces of information. Note that the storage scheme used by sockets allows them to compact data of the same type into the minimal number of mbufs required to hold those data.

On entry to *soreceive()*, a check is made to see whether **out-of-band** data are being requested. If they are, the protocol layer is queried to see **whether** any such

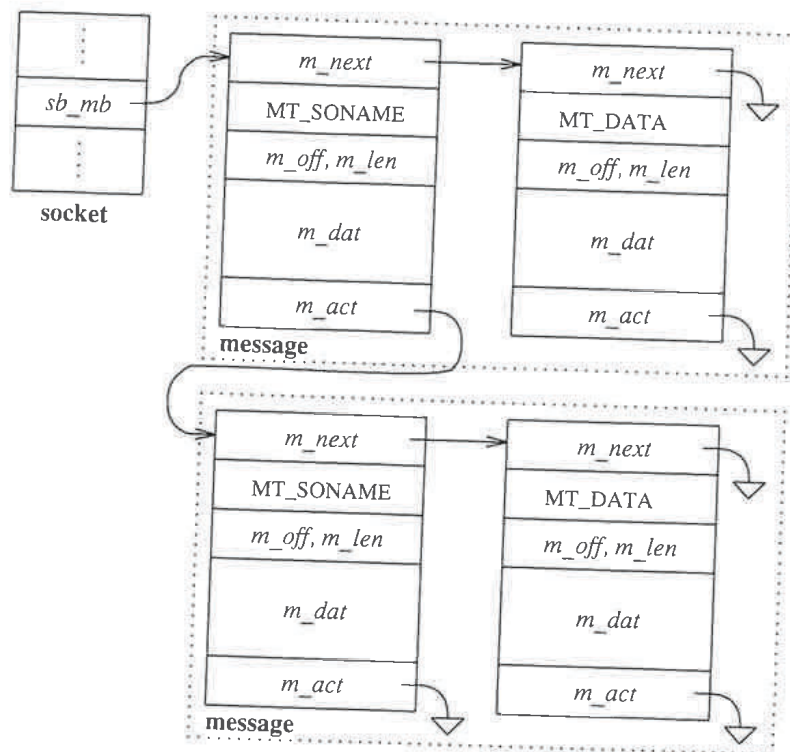


Figure 10.10 Data queuing for datagram socket.

data are available; if the data are available, they are returned to the caller. As regular data cannot be retrieved simultaneously with out-of-band data, *soreceive()* then returns. Otherwise, regular data have been requested, and *soreceive()* checks the receive data buffer character count to see whether data are available. If no data are present, the socket's state is consulted to find out whether data might be forthcoming. Data may no longer be received, because the socket is disconnected (and a connection is required to receive data), or because the reception of data has been terminated with a *shutdown* by the socket's peer. In addition, if an error from a previous operation was detected asynchronously, the error needs to be returned to the user; *soreceive()* checks the *so_error* field after checking for data. If no data or error exist, data might still arrive, and if the socket is not marked for nonblocking I/O, *soreceive()* puts the process to sleep to await the arrival of new data.

When data arrive for a socket, the supporting protocol notifies the socket layer by calling *sorwakeup()*. The contents of the receive buffer can then be processed, observing the data-structuring rules described previously. *Soreceive()* first removes any address that must be present, then optional access rights, and finally normal data. The removal of data is slightly complicated by the interaction between in-band and out-of-band data. The location of the next out-of-band datum is marked in the in-band data stream and is used as a record boundary

during in-band data processing. That is, when an indication of out-of-band data is received by a protocol, the corresponding point in the in-band data stream is marked. Then, when a request is made to receive in-band data, only data up to the mark will be returned. This mark allows applications to synchronize the in-band and out-of-band data streams so that, for example, received data can be flushed up to the point at which out-of-band data are received. Each socket has a field, *so_oobmark*, that contains the character offset from the front of the receive data buffer to the point in the data stream at which the last out-of-band message was received. When in-band data are removed from the receive buffer, the offset is updated accordingly, so that data past the mark will not be mixed with data preceding the mark. The *SS_RCVATMARK* bit in a socket's state field is set when *so_oobmark* reaches zero to show that in-band and out-of-band data streams are synchronized. An application can test the state of this bit with the *SIOCATMARK* *ioctl* call to find out whether all in-band data have been read up to the point of the mark.

Once data have been removed from a socket's receive buffer, *soreceive()* updates the state of the socket and notifies the protocol layer that data have been received by the user. The protocol layer can use this information to release internal resources, to trigger end-to-end acknowledgment of data reception, to update flow-control information, or to start a new data transfer. Finally, if any access rights were received, *soreceive()* passes them to a communication domain-specific routine to convert them from their internal representation to the external representation.

Passing Access Rights

In addition to the transmission and reception of uninterpreted data, the system also supports the passage of data that have special meaning. These data normally represent the right to perform operations on associated objects. The data used to represent access rights, or capabilities, normally are meaningful only within the context of the process that created or obtained the right, so their transmission requires system support to make them meaningful in a receiving process's context. For example, in UNIX, access rights to files in the filesystem or sockets are encapsulated in the form of file descriptors. A file descriptor is a small integer number that is meaningful only in the context of the process that opened or created the associated file. To pass a file descriptor from one process to another, the system must create a reference to the associated file-table structure in the receiving process's user structure.

Access rights, or capabilities, are categorized as *internalized* or *externalized*. Internalized capabilities require the support of trusted agents to be useful. Keys associated with these capabilities are created by a trusted agent, and, when presented for the purpose of accessing a protected object, are deemed valid according to their interpretation in the context of the presenter.

Externalized capabilities, on the other hand, utilize keys that require no specific trusted agent for their use. That is, the validation of the right to access an object is based solely on the possession and presentation of the requisite key.

Systems that use externalized capabilities frequently utilize a public key-encryption algorithm. Keys for externalized capabilities normally have the property that they are long-lived and that they may be stored in locations such as a filesystem without losing their usefulness.

No specific system support is required to support externalized capabilities. To support internalized capabilities, however, the operating system, acting as a trusted agent, must verify and translate keys when transmitting them as messages between processes. The interprocess-communication system provides facilities, on a per-communication domain basis, to process all access rights transmitted and received in messages, and to dispose of rights that are not received.

Sending and receiving of access rights requires the internalization and externalization of these rights. Internalization converts a key held by a sending process into an internal form that can be passed as data in a message. Externalization reverses this process, converting the internal form into an external form that is meaningful in the context of the receiving process. Internalization of access rights is performed at the protocol layer when the *send()* routine requests transmission of data containing access rights. The access rights to be transmitted are passed as an mbuf chain separate from the regular data. When *receive()* encounters access rights on the receive data queue, it invokes the communication domain's *dom_externalize* routine to externalize the rights. The socket layer implicitly presumes that access rights stored in socket data queues will be valid as long as the system remains up. That is, there are no mechanisms to expedite the delivery of access rights, or to time out or invalidate rights stored on a socket data queue.

Access Rights in the UNIX Domain

In the UNIX domain, the internalization of file descriptors results in their conversion to system file-table pointers, whereas externalization requires allocation of new file descriptors for the receiving process. File descriptors passed in messages are actually duplicates of the ones held by the sending process (as though they had been created by *dup*). The sending process must explicitly close a file descriptor after that descriptor has been sent, in order to give it away.

A *garbage-collection* facility is provided to reclaim resources associated with access rights that are not delivered properly. Access rights may not be delivered for two reasons: either because the receiving socket has insufficient space, or because the user does not request them with the proper system call when receiving data from the socket. Garbage collection is used because normal message processing does not permit a protocol to access a message after it has passed on that message for delivery. This inability to access a message after it has been transmitted means that, if access rights in a message are not delivered, these rights will be discarded without being reclaimed.² In the UNIX domain, reclamation of access rights ensures that files associated with these rights are closed so that

² With hindsight, this problem would have been better handled with use of the per-domain disposal routine during freeing of an mbuf typed with *MT_RIGHTS*.

system resources such as file-table entries are not depleted.

For garbage collection to be implemented, each file-table entry must contain a count of references held by file descriptors present in socket receive queues, *f_msgcount*. Another variable, *unp_rights*, tracks the number of file descriptors held in all the UNIX domain sockets in use. When a file descriptor is internalized to a file-table pointer for the purpose of transmission, the *f_msgcount* for the file is incremented. On reception, when the file descriptor is externalized, *f_msgcount* is decremented. When a UNIX domain socket is reclaimed and *unp_rights* is nonzero, the garbage-collection routine, *unp_gc()*, is invoked to scan the file table and all UNIX domain sockets to reclaim unaccounted-for file-table references.

Unp_gc() uses a *mark-and-sweep algorithm* in performing its duties [Cohen, 1981]. The basic strategy is to locate all references to files that are present in UNIX domain sockets or in a process's user structure. Files that have existing references either in a user structure or in a socket's receive data buffer are *marked*, whereas those that are *thought* to be active, but for which no existing reference can be found, are not *marked*. The garbage collector can then reclaim lost references by searching the file table for *unmarked* entries for which all references are indicated as being in socket receive queues.

Note that the garbage collector is invoked only when a UNIX domain socket is closed and file descriptors are known to be queued awaiting reception; thus, the overhead associated with the garbage collector is limited. Also, the garbage collector reclaims only those file-table entries that were lost while being passed in messages; references that might be lost in other parts of the system are not reclaimed.

10.7 Socket Shutdown

Although closing a socket and reclaiming its resources at first glance appears to be a straightforward operation, it can in fact be rather complicated. The complexity arises because of the implicit semantics UNIX attaches to the *close* system call. In certain situations (e.g., when a process exits), a *close* call is never expected to fail. However, when a socket promising reliable delivery of data is closed with data still queued for transmission or awaiting acknowledgment of reception, the socket must attempt to transmit the data, perhaps indefinitely, in order for the *close* call to maintain the socket's advertised semantics. If the socket discards the queued data to allow the *close* to complete successfully, it violates its promise to deliver data reliably. Discarding data can cause naive UNIX processes, which depend on the implicit semantics of *close*, to work unreliably in a network environment. However, if sockets block until all data have been transmitted successfully, in some communication domains a *close* may never complete!

The socket layer compromises in an effort to address this problem yet to maintain the semantics of the UNIX *close* system call. Figure 10.11 shows the possible state transitions for a socket from a connected to a closed state. In normal operation, closing a socket causes any queued but unaccepted connections to

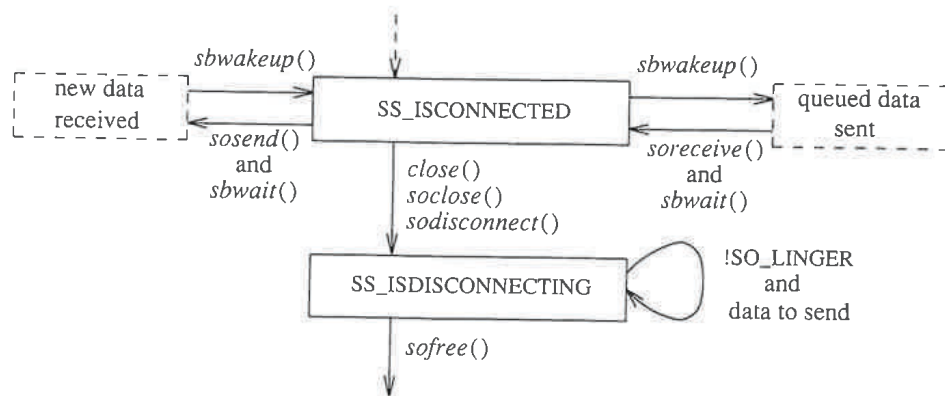


Figure 10.11 Socket-state transitions during shutdown.

be discarded. If the socket is in a connected state, a disconnect is initiated. The socket is marked to indicate that a file descriptor is no longer referencing it, and the close operation returns successfully. When the disconnect request completes, the network support notifies the socket layer, and the socket resources are reclaimed. The network layer may attempt to transmit any data queued in the socket's send buffer, although this is not guaranteed.

Alternatively, a socket may be marked explicitly to force the application process to *linger* when closing until pending data have drained and the connection has shut down. This option is marked in the *socket* data structure using the *setsockopt* system call with the *SO_LINGER* option. When an application indicates that a socket is to linger, it also specifies a duration for the lingering period. If the lingering period expires before the disconnect is completed, the socket layer forcibly shuts down the socket, discarding any data still pending.³

Exercises

- 10.1 What limitation in the use of pipes inspired the developers to design alternative interprocess-communication facilities?
- 10.2 Why are the 4.3BSD interprocess-communication facilities designed to be independent of the UNIX filesystem for naming sockets?
- 10.3 Why is interprocess communication layered on top of networking in 4.3BSD, rather than the other way around?

³ Actually, lingering is only partly implemented; the lingering period is infinite and cannot be changed by users.

- 10.4 Would a screen editor be considered a naive or a sophisticated UNIX program, according to the definitions given in this chapter? Explain your answer.
- 10.5 What are out-of-band data? What type of socket supports the communication of out-of-band data? Describe one use for out-of-band data.
- 10.6 Give two requirements that interprocess communication places on a memory-management facility.
- 10.7 How many mbufs would be needed to hold a 3024-byte message? Draw a picture of the necessary mbuf chain and any associated memory pages.
- 10.8 Why does an mbuf have two link pointers? For what is each pointer used?
- 10.9 Why is the 4.3BSD socket data structure limited in size to at most 112 bytes?
- 10.10 Each socket's send and receive data buffers have high and low watermarks. For what are these watermarks used?
- 10.11 Consider a socket with a network connection that is queued at the socket awaiting an *accept* system call. Is this socket on the queue headed by the *so_q* or by the *so_q0* field in the socket structure? What is the use of the queue the socket is *not* on?
- 10.12 How does the protocol layer communicate an asynchronous error to the socket layer?
- 10.13 Sockets explicitly refrain from interpreting the data that they send and receive. Do you believe that this is the correct approach? Explain your answer.
- 10.14 Why does the *sosend()* routine ensure there is enough space in a socket's send buffer before making a call to the protocol layer to transmit data?
- 10.15 How is the type information in each mbuf used in the queueing of data at a datagram socket? How is this information used in the queueing of data at a stream socket?
- 10.16 Why does the *soreceive()* routine optionally notify the protocol layer when data are removed from a socket's receive buffer?
- 10.17 Describe an application where the ability to pass file descriptors is useful. Is there another way to simulate this facility in 4.3BSD?
- 10.18 What is the difference between an internalized capability and an externalized capability? Would UNIX descriptors be considered externalized or internalized capabilities according to the definitions given in this chapter?
- 10.19 What might cause a connection to linger forever when closing?

- *10.20 What effect might storage compaction have on the performance of network-communication protocols?
- **10.21 Why is releasing mbuf storage back to the system complicated? Is it desirable?
- **10.22 In the original design of the interprocess-communication facilities, a reference to a communication domain was obtained with a *domain* system call,

```
int d; d = domain("inet");
```

(where *d* is a descriptor, much like a file descriptor), and sockets then were created with

```
s = socket(type, d, protocol);
int s, type, protocol;
```

What advantages does this scheme have over the one that is used in 4.3BSD? What effect does the introduction of a domain descriptor type have on the management and use of descriptors within the kernel?

References

- Cerf, 1978.
V. Cerf, "The Catenet Model for Internetworking," Technical Report IEN 48, SRI Network Information Center, Menlo Park, CA (July 1978).
- Cohen, 1977.
D. Cohen, "Network Control Protocol (NCP) Software," University of Illinois Software Distribution, University of Illinois, Champaign-Urbana, IL (1977).
- Cohen, 1981.
J. Cohen, "Garbage Collection of Linked Data Structures," *Computing Surveys* 13(3), pp. 341-367 (September 1981).
- Fitzgerald & Rashid, 1986.
R. Fitzgerald & R. F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent," *ACM Transactions on Computer Systems* 4(2), pp. 147-177 (May 1986).
- Gurwitz, 1981.
R. F. Gurwitz, "VAX-UNIX Networking Support Project—Implementation Description," Technical Report IEN 168, SRI Network Information Center, Menlo Park, CA (January 1981).
- Kalash *et al.*, 1986.
J. Kalash, L. Rodgin, Z. Fong, & J. Anton, "Ingres Version 8 Reference Manual," pp. 10:1-88 in *UNIX Programmer's Supplementary Documents*,